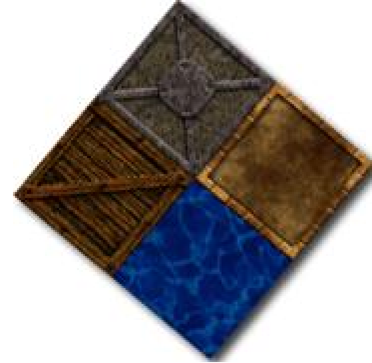
Torque Game Builder – Image Maps

Introduction

This document is intended to give you a complete overview of Torque Game Builder's (TGB) Image-Map system from what Image-Maps are to how they are typically used so without delay, let's begin.



One of the big hurdles when developing any product is managing resources and there are few types of resources more troublesome than images. Not only do images occupy lots of disk-space and memory but they are typically numerous. Add to this that there are performance issues related to the way images are stored/accessed and that these images have various runtime options associated with them and the situation can get quite complex.

Here's where TGB provides a helping hand; enter the '*t2dImageMapDatablock'.* As the name implies, the '*t2dImageMapDatablock'* is a type of datablock hereto referred to as an Image-Map.

The first question here is naturally, "what is a datablock?" and although a full discussion of the capabilities of datablocks is beyond this document, it's worth providing a very quick overview.

Datablocks are specialized objects that provide a portable/reusable container for common and typically-static 'data'. Data here could refer to characteristics of an enemy's movement, the look/feel of GUI elements or more importantly for this document, definition of your imagery.

An example of a datablock would be one that defines the properties of an enemy so that whenever an enemy is created, it takes its start-up properties from the datablock. By changing the datablock definition, you can globally change the behavior of your enemies.

Typically, datablocks do not change at runtime but there is absolutely nothing to stop you doing this. Some things can be changed dynamically, some can't.

The great thing about datablocks is that they have their own keyword within the Torque-Script language call "datablock" in the same sense that a function has, err … well, "function" but we'll get into the details of configuring datablocks shortly.

The most important way to think about datablocks are that they are objects that allow you to group logical, common data for groups of objects to use allowing you to make broad changes to any aspect of your game whether that be the look/feel or even the behaviors of game elements such as enemies or weapons.

Well this is all rather vague so let's get into something more practical; let's start defining some real datablocks and see how they work.

Frame Configuration

So now we've provided a basic idea of what a datablock is, let's breakdown what exactly Image-Maps allow you to do. The idea here is to specify an image, how the frames within this image are organized and to provide various runtime options. The basic format for an Image-Map is:

```
datablock t2dImageMapDatablock( myImageMap )
{
};
```

Note that you define a datablock by using the keyword "datablock" followed by the type of datablock you want to define. In this case, it's a 't2dImageMapDatablock', a type which we simply call "Image-Maps". The final part of the definition is the name, here we're calling it "myImageMap" but you would probably call it something more related to its usage, perhaps "playerShip" or "bigFont".

Now that we've got the basic definition of Image-Maps, let's start populating the datablock with useful information. We do this by adding a datablock "field" to the standard datablock definition. A field has the following format:

```
datablock t2dImageMapDatablock( myImageMap )
{
   myField = "foobar";
};
```

The above example isn't that useful but what it does do is create a field called "myField" and assigns it the string "foobar". To access this field we'd use something like:

```
echo( myImageMap.myField );
```

This shouldn't be difficult to understand as this dynamic-field creation is just an extension of standard torque-script you've probably already been using. So now that we know we can put our own custom-fields into our datablocks, let's look at the very important predefined ones that drive Image-Maps:

- “imageName” - Source Imagery
- “imageMode” - Previously called the ambiguous "mode"
- “frameCount” - Optional; specifies the expected frame-count.
- “cellCountX” - *CELL mode only*; 'horizontal' span of cells.
- “cellCountY” - *CELL mode only*; 'vertical' span of cells.
- “cellWidth” - *CELL mode only*; size of 'horizontal' cells.
- “cellHeight” - *CELL mode only*; size of 'vertical' cells.
- “cellOffsetX” – *CELL mode only*; ‘horizontal’ pixel-offset to start of tile capture.
- “cellOffsetY” – *CELL mode only*; ‘vertical; pixel-offset to start of tile capture.
- “cellStrideX” - *CELL mode only*; ‘horizontal’ pixel-stride during tile capture.
- “cellStrideY” - *CELL mode only*; ‘vertical’ pixel-stride during tile capture.
- “cellRowOrder” - *CELL mode only*; Specifies if row-order is used during tile capture.
- “linkImageMaps” – *LINK mode only;* Specifies the list of image-maps to link together.
- “filterMode” - Rendering control of the GPU filter mode.
- “filterPad” - GPU filtering artifact fix.
- “preferPerf” – Prefer Performance over least-waste.
- “preload” – Preload Textures when created. Texture(s) are always active!
- “allowUnload” – Specifies whether the texture(s) can unload.

This list of fields may at first seem daunting but bear in mind that most of the time you’ll only uses a couple of them, a handful are for more advanced control of image-maps only. Okay, let's go through each of these fields and expose how indeed, Image-Maps do provide plenty of flexibility in getting your artwork into TGB.

"imageName"

This field defines where the source image is located on the disk. An important thing to note here is the use of the term "image". When we say "image", we are referring to a bitmap on the disk that contains imagery. Now although this sounds like an overstatement, it's important in understanding what a "frame" is. Quite often, developers use a single "image" that contains multiple "frames".

Here's an example of a typical image from the SDK that contains multiple frames:

As you can see, this single image contains four separate frames. The image size is 256 wide by 256 high but each frame is 128 wide by 128 high. We'll use this image for some examples so you may want to note these dimensions so that as we discuss frames you'll have the dimensions to hand.

Multiple frames are great but why would we make it so complicated by combining frames when we could've easily created four separate images? Well, the answer is essentially simple but hides complex inner workings of your graphics-card. The combining of frames into a single image reduces not only the overhead of storing potentially hundreds of separate files but also loading/interpreting them. A more complex reason is due to the way that graphics hardware works (*see technical note at the end of this "imageName" section*).

So let's specify the image above in our example image-map:

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/tileMap";
};
```

Here we've told TGB that we've got a bitmap located in the directory *"t2d/data/images/"* and its name is *"tileMap"*. We've highlighted this field in bold to indicate that you must always specify this field in an image-map. Some fields are optional and we'll not highlighted those. Note also that we purposely omitted the files' extension as TGB will attempt to load the different formats that it supports automatically, in a specific order as follows:

	24/32-Bit Images (Typical)	8-Bit (Palletized) Images
1st	**.jpg*	**.bm8*
2nd	**.png*	**.bmp*
3rd	**.gif*	**.jpg*
4th	**.bmp*	**.png*
5th	-	**.jpg*

Non-palletized 24-bit images can get their alpha-layer automatically loaded from a separate ".jpg" file if it is alongside the specified one and it contains a suffix of ".alpha.jpg"

******TECHNICAL NOTE ******

When TGB uploads your imagery into the graphics card, it attempts to place them into the same graphics object called a "texture". Current graphics cards have limitations on the dimensions of these texture objects. Older cards can only handle 256 wide by 256 high textures whereas more modern cards can handle up to 4096 by 4096.

There's also a limitation (on some cards) that the dimensions of these texture-objects have to be a power-of-two in size e.g. 2,4,8,16,32,64,128,256,1024,2048,4096 etc.

Another complexity is that when TGB is rendering your sprites, it needs to tell the graphics card which texture it requires and the graphics-card selects the appropriate texture accordingly. This is no problem until TGB is selecting frame after frame which all reside on different textures. Because the graphics-card can only select one texture-object at a time, this can lead to the graphics-card swapping between different texture-objects continuously. This is commonly called "texture thrashing" and generally reduces performance, sometimes quite considerably. TGB objects such as the t2dTileLayer can cause this as it renders lots of tiles (therefore frames) during a single screen-update.

There are a couple of ways counteracting this problem; rendering the objects in a specific order so that all objects that use the same image are rendered together or alternatively, try to keep as many frames in as few textures as possible. Also, grouping similar frames into the same texture can be of benefit.

The good news is that TGB has an intelligent packing routine that will take the frames from your bitmap images that you supply, reorganize them internally to ensure that the texture card not only gets textures that it can handle but also to try to produce the best performance texture-objects that it can, totally automatically. Put another way; unlike the older versions of TGB (v1.0.2-), your input bitmap sizes DO NOT have to meet the above conditions. TGB will extract the frames and reorganize them into compatible textures. Feel free to create bitmaps that you're happy with!

The only exception is that no individual frame can be larger than that supported by the graphics hardware. In the future, even this condition may be removed with TGB automatically splitting-up the frame into manageable chunks allowing TGB to render it seamlessly albeit at a loss in performance.

More details on this later.

“imageMode” (previously called just “mode”)

So now we've told the image-map which image contains the frames we're interested in but it doesn't know where the frames are within the image so we need to define those. This is done by defining a specific Image-Mode. The concept of an Image-Mode is probably the most complex part of an Image-Map so as soon as you've mastered this field, you're pretty much ready to go.

There are four types of image-modes that you can use, these are:

- `FULL – Full-Image (Single-Frame).`
- `CELL – Regularly-Spaced (Multiple-Frames).`
- `KEY – Color-Keyed (Multiple-Frames).`
- `LINK – Linked Image-Map (Multiple-Frames).`

These four types of image-modes allow you to extract frames from your imagery in different ways. To some degree, TGB is flexible on the arrangement of frames within images but you'll still need to be careful to arrange your frames so that you get expected results.

Let's go through each of these image-modes in turn.

"FULL" Image-Mode

This mode is extremely simple to understand as it simply tells TGB that the whole image is a single frame. There are no extra parameters you need to specify at all! It's probably a good time to mention that TGB will assign any frames it finds, a single number starting at zero. In this mode, you'll get a single frame referred to as frame 0 (zero).

Here's an example datablock defining a *"full"* image-mode image-map:

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/bigMap"
   imageMode = full;
};
```

As you can see, we simply add the field "imageMode" and set it to "full" and we're done. Note that this field is highlighted bold here which means you must always specify this field.

"CELL" Image-Mode
This mode is more complex but still easy to understand once you've used it a few times. It's also the "bread and butter" image-mode that you'll more than likely be using all the time so it's important to go through this particular mode in detail. Because this mode is the most common one used, it's also the default that if you don't actually specify "imageMode", TGB will automatically assume you want *"cell"* mode.

In *"cell"* mode, you're telling TGB that your frames are organized into a regular grid where all frames have a common width/height, each frame being termed a *"cell"*.

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/tileMap"
   imageMode = cell;
   cellWidth = 128;
   cellHeight = 128;
   cellCountX = 2;
   cellCountY = 2;
};
```

As you can see, our image-map has suddenly expanded with lots of extra fields. Well, don't worry, everything will become clear soon. We've added the *"imageMode"* as *"cell"* as discussed above but we've also added four more fields described briefly at the beginning of this topic. The reason we're adding them now is because these fields are only used when you're in *"cell"* mode. Let's talk about these extra fields.

In *"cell"* mode, TGB scans across your image from top-left until it reaches the right of your image, it then moves down a *"cell"* and repeats until it reaches the bottom of your image. To do this, TGB needs at least two pieces of information, these are the *"cellWidth"* and the *"cellHeight"*. Given these two fields, TGB can successfully capture frames from the image. As you may notice these fields must be specified but only if the *"imageMode"* is *"cell"*. So what about the two other fields *"cellCountX"* and *"cellCountY"?* Well these are ***optional*** fields that allow you to restrict how many cells horizontally and vertically you want from the starting position of top-left. If you omit either of these fields, TGB will try to get as many of these cells in the respective direction as it can.

So if we look at the example image-map datablock we've provided, we can see that we've told TGB that we expect a bunch of cells that are 128 wide by 128 high and that we expect 2 cells horizontally and 2 cells vertically. Of course, we didn't need to explicitly state that we wanted two cells horizontally and vertically, we could've just specified the following:

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/tileMap"
   imageMode = cell;
   cellWidth = 128;
   cellHeight = 128;
};
```

This would've given us exactly the same results because as TGB scans across the image horizontally, it only finds room for two 128 wide cells (frames 0 and 1) before it hits the edge of the image so it moves down by the *"cellHeight"* and scans another row. It then finds another two cells (frames 2 and 3) before it again hits the edge of the image so it moves down by the *"cellHeight"* again and hits the bottom of the image so it quits. At this point, it's found 4 frames (frames 0, 1, 2 and 3).

When you're using optional fields, such as *"cellCountX"* or *"cellCountY"*, TGB does not force you to use both at the same time. It's quite happy if you only want to specify the Y count and leave the X count for it to determine. **This goes for all optional field pairs!**

Let's say, for some reason, that we only want to get two frames at the left of the image (the stone tile and the crate tile). What we need to do here is simply tell the image-map that we only want a single tile in the horizontal direction but two in the vertical like so:

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/tileMap"
   imageMode = cell;
   cellWidth = 128;
   cellHeight = 128;
   cellCountX = 1;
   cellCountY = 2;
};
```

If you're pretty happy with what you've seen so far, it might be worth you trying out both the *"FULL"* image-mode and the *"CELL"* image-mode before we go further. Come back when you're ready to proceed, we'll still be here!

You back already?

Okay, let's continue discussing the *"CELL"* image-mode now that you're more familiar with the basics.

Now that you've used this mode a little, you may have noticed a few restrictions when extracting frames from your images. To start with, it looks like you're restricted to starting at the top-left of your image. Also, frames are extracted rightwards and downwards. Well, as you would expect, TGB doesn't restrict you like this at all. Let's introduce a couple of the remainder of the *"CELL"* image-mode fields:

- "cellOffsetX" – *CELL mode only*; 'horizontal' pixel-offset to start of tile capture.
- "cellOffsetY" – *CELL mode only*; 'vertical; pixel-offset to start of tile capture.

Take the first restriction we spoke about which was that the frame extraction always starts at the top-left of your image. Well this isn't so! If you don't specify either *"cellOffsetX" or "cellOffsetY"* then indeed, frame extraction will start at the top-left of your image. In-fact, these cell-offsets both default to zero so that's why TGB starts at the top-left! If you want to start at a specific position within your image, you can specify how many horizontal/vertical pixels you want with *"cellOffsetX" or "cellOffsetY"*. Let's show an example to see this working:

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/tileMap"
   imageMode = cell;
   cellWidth = 128;
   cellHeight = 128;
   cellOffsetX = 128;
   cellOffsetY = 0;
   cellCountX = 1;
   cellCountY = 2;
};
```

What we did here was to specify the offset as 128 pixels horizontally and 0 pixels vertically which equates to the pixel at the top-left of the region marked "0" above. In other words, the top-left of where our frame extraction will begin. If we look at the other fields we specified told TGB that there is 1 cell horizontally, 2 cells vertically and that the cells are 128 wide by 128 high each. TGB will start at the offset position and extract the first frame 0. It will then move right by the cell-width where it hits the right of the image so it moves down by the cell-height back to the specified *"cellOffsetX"* where it extracts frame 1. It then moves right by the cell-width, hits the right of the image, moves down by the cell-height but hits the bottom of the image so it finishes. The result; two frames (0 and 1).

So the rule is that cell-offsets specify the top-left position of your starting frame.

As we mentioned before, some of these fields are optional meaning that TGB can still make sense of what it thinks you want without them but it may sometimes get it wrong if you want something special.

In the datablock above, we specify the *"cellOffsetY"* as 0 but it defaults to that so we didn't really need to do it. Also, we set *"cellCountX"* to 1 but TGB will already determine that it can only extract a single frame horizontally from the specified offset. So why did we do it? Well, it's always a good thing to make your intent clear, especially as other programmers/artists may not know what the defaults are and you may yourself forget at a later date. This is not a rule, that's why these fields are totally optional, just good advice!

Well, we've covered the cell-offsets so let's move onto the final set of fields that might at first be a little confusing. Let's step back to one of the restrictions we spoke about earlier. We stated that all frames are extracted rightwards and downwards. Indeed all our descriptions so far have been based upon the extraction routine moving right, hitting the edge of your image and then moving down with the extraction process stopping when it hits the bottom of your image.

Well, it shouldn't be surprising to know that TGB allows you to specify how it moves to the next frame. Let's introduce nearly the remainder of the *"CELL"* image-mode fields:

- "cellStrideX" - *CELL mode only*; 'horizontal' pixel-stride during tile capture.
- "cellStrideY" - *CELL mode only*; 'vertical' pixel-stride during tile capture.

If you don't specify either of these fields, TGB will extract frames as we've so far specified moving rightwards and downwards. What these fields allow you to do is override this movement! Okay, say we've gone crazy and we want to extract some frames in very specific order. Again, we'll use the example image but this time, we want to extract our frames starting at the bottom-right frame, moving leftwards and upwards which is completely reverse to what TGB does by default. Here goes:

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/tileMap"
   imageMode = cell;
   cellWidth = 128;
   cellHeight = 128;
   cellOffsetX = 128;
   cellOffsetY = 128;
   cellStrideX = -128;
   cellStrideY = -128;
   cellCountX = 2;
   cellCountY = 2;
};
```

As you can see, our frames are now in the reverse-order to what we originally had. All we did to achieve this was to specify the cell "stride". The "stride" is the movement that occurs after each frame is extracted. Let's go through what we did above.

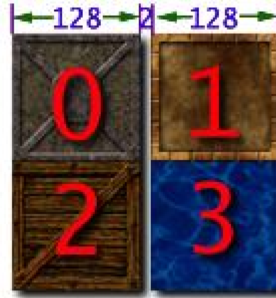
First, you'll note that we specified the offset to be (128 horizontally by 128 vertically) which equates to the top-left of the frame marked "0" above. **You must always use the top-left of your starting frame for your offset!** The new part is that we've specified both a horizontal and vertical stride. These are both -128. Why the minus? Well, as you've probably already worked-out, minus for the horizontal means "move left" whilst minus vertical means "move up". Positive values for these strides mean "right" and "down" respectively.

So what TGB did was to start at the frame marked "0" because we specified an offset of (128,128), it then extracted that frame and because we specified a horizontal stride, it moved that stride which is (-128) which placed us at the top-left of the frame marked "1" where it extracted that frame. It then moved the horizontal stride again but hit the left of the image so TGB moved by the vertical stride which is (-128) and reset the position to the *"cellOffsetX"* which placed it at the frame marked "2" where it extracted that frame. Finally, it moved the horizontal stride again which placed it at the final frame marked "3". After extracting this frame, it moved by the horizontal stride which hit the left of the image so TGB moved by the vertical stride which also hit the top of the image causing the image extraction to finish.

As you can see, cell-strides allow you to change the direction of the image-extraction. This combined with the ability to specify a starting position using the cell-offsets provides great flexibility.

Okay, now that we've covered using cell-strides, let's talk about using them for a possibly unexpected reason. Say that our example imageMap had frames that were still 128 wide by 128 high but that they had "gaps" between the frames that we don't want to acquire; how do we get the frames but ignore the gaps? Here's how:-

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/tileMap"
   imageMode = cell;
   cellWidth = 128;
   cellHeight = 128;
   cellStrideX = 130;
   cellCountX = 2;
   cellCountY = 2;
};
```



First, let's describe the example image; the image has four frames, each frame being 128 wide by 128 high as before but now there is a gap of two pixels horizontally between the frames making the total image-width 128+2+128 = 258 and the total image-height 128+128 = 256 as before.

So how do we tell TGB so skip that annoying gap? Well, it's actually really easy as it's just an extension of what you've already seen. Notice we didn't specify the cell-offset so TGB will start at the top-left of our image which is just fine with us. As you know, when TGB has extracted the first frame marked "0" above, it will either move horizontally by the cell-width or, if we've specified, move horizontally by the *"cellStrideX"*. As you can see, we did specify it as 130. What TGB will obviously now do is to step 130 pixels horizontally to the right to the next frame which positions us at the frame marked "1". We successfully bridged the gap! The rest is quite easy to understand as we've done it before. When TGB hits the right of the image, it will move down either by the cell-height or, if we specified, the *"cellStrideY"*. Well we didn't specify it in this example so TGB moves down by the cell-height which takes us to the frame marked "2". The final stage is that TGB moves horizontally by our specified *"cellStrideX"* of 130 which takes us to our final frame marked "3".

As you can see, strides not only allow you to extract frames in different directions but also allow you to skip regular spacing in your images. Quite often, older "color-keyed" images surround each frame with a specific colour and you can use your offsets/strides to skip these borders and directly extract the frames.

Phew! As you can see, there's lots of different ways you can extract stuff in *"cell"* mode and thankfully, we've only got a single field to go and then you'll know all there is to know about this mode.

The final field is:

- `"cellRowOrder" - CELL mode only; Specifies if row-order is used during tile capture.`

It shouldn't be too much trouble understanding this one as it's really just a change of direction. As we've mentioned in all the examples so far, TGB is very flexible on how it extracts frames using offsets/strides/widths and heights. The one thing, that you may have noticed, that has stayed the same during all these frame extraction examples is the fact that T2D always moves horizontally until it hits either the left or right of your image and then vertically until it hits either the top or bottom of your image.

In-fact, as you may have guessed, this is only the default and can again be changed. The field *"cellRowOrder"* is simply a flag either *true* or *false* (defaults to *true*) that specifies if cells are to be extracted in row-order or alternately column-order so *true* = row-order and *false* = column-order.

For our final example in *"cell"* mode, let's use the bog-standard example we used in the beginning but let's extract the images in the non-standard column-mode like so:

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/tileMap"
   imageMode = cell;
   cellWidth = 128;
   cellHeight = 128;
   cellRowOrder = false;
};
```

As you can see, instead of the frames being extracted horizontally, they've been extracted vertically simply by changing this field. Note that you can still use the offsets/strides as explained previously; the only change is that frames are extracted columns first rather than rows, you should still get the same number of frames extracted, just in a different order.

Although this offers limited practical value, it does allow a preferred style of storing your frames or even the flexibility if someone provides you with frames in this order.

Okay, I think you can give yourself a pat on the back here as you've now completely covered *"cell"* mode. Go away and practice with it until you're totally happy that it does what you want. Come back here when you're ready to continue our tour of TGB image-maps.

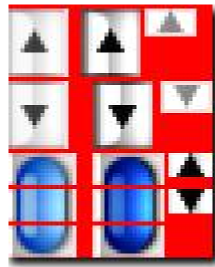
"KEY" Image-Mode

Welcome back! Well, so far you should know how to setup an image-map to get a single full-frame using *"FULL"* image-mode as well as creating multiple, regularly-spaced, same-sized frames using *"CELL"* mode but what if we want multiple-frames of different sizes and they're not regularly-spaced in our image? Well, TGB provides another mode to allow you to do just that, the *"KEY"* image-mode.

In this mode, you can place your frames with a large degree of freedom. The frames can also be of different sizes. So how does TGB know where the frames are? Well, it uses a technique called color-keying. Color-keying is a technique used for special-effects in the movies albeit TGB uses it slightly differently. In the movies, a specific color is used in the scene to identify the "background" so that nice scenes can be added later. If you've seen actors working against blue or green backgrounds, you've seen color-keying. Unlike in the movies, we're not interested in the background; we're **not** interested in the actors e.g. the frames.

This *"key*" mode can at first be difficult to understand but there only a few rules you need to be aware of. As motivation in understanding this mode, it's important to know that the image-format used here is identical to the one used when skinning GUI elements in TGB so by understanding this, you'll also understand how to format your images to skin your GUI!

Okay then, let's start with an image that's been setup to have its frames extracted using the *"key*" image-mode:



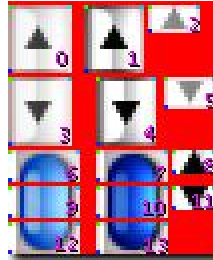
As you can see, there are 14 frames of various sizes placed in non-regular positions. So how does TGB extract the frames and in what order? Well, TGB uses simple rules to achieve this and although it looks like the frames have been placed arbitrarily, they've actually been placed according to these rules. Let's start by explaining the steps that TGB goes through to find frames. Note that unlike the *"cell*" image-mode, frame-extraction is always fixed moving rightwards and downwards.

The first step TGB takes before looking for frames is to choose the color-key. It does this by simply using the color at location (0, 0) e.g. the top-left pixel of the image. As you can see, this is red. From now on, TGB assumes that everything that is this color is background only and nothing to do with frames. It is for this reason that that it's extremely important that you choose a color that isn't used by your frames! You can choose a different color for each image-map but the same one must be used within each individual image.

Torque Game Builder – Image Maps

Okay, now that TGB has the color-key, it can begin looking for frames. To make this explanation a little easier, here's the same image and its image-map definition but we've marked areas that we're going to discuss as well as showing the order of frame-extraction that TGB uses:

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/guiMap"
   imageMode = key;
};
```



The first step it takes is to determine a row to scan across to look for the top-left of your frames. It does this by scanning down the left-most column for a color that ***isn't*** the color-key. As you can see, if it does this, it'll encounter the top-left of frame 0 (marked with the green pixel). When it's found this, TGB then scans along the row to the right until it finds the color-key. As you can see, this will be the cyan pixel in frame 0. Now TGB knows the top-left of the frame and its width so it only needs to determine the frames height and it can extract the frame. It does this by scanning-down from the top-left of the frame (marked with a green pixel) until it reaches the bottom-left of the frame 0 (marked with a blue pixel). Now TGB has the top-left, width and height of the frame so it extracts it!

What happens next is very important. TGB starts scanning from just past the top-right of the last frame (marked with a cyan pixel) until it reaches a non-color-key pixel. When it does this, it finds the top-left of frame 1 (marked with a green pixel). It continues to find the width and height of this frame exactly as it did the previous frame. TGB then extracts the frame and repeats finding frames until it hits the right-side of the image.

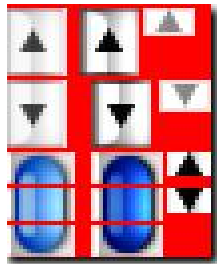
Okay, let's continue to extract more frames. After TGB has extracted the first 3 frames and has hit the right-side of the image, it moves straight back to the left-side of the image and continues scanning for more frames. The next frame it encounters will be the top-left of frame 3 (marked with a green pixel). TGB then repeats the procedure outline above. TGB will continue extracting frames until it hits the bottom of the image.

Before we go any further, let's show some of the rules with what we've learned so far. First, it should be obvious that you need to put your color-key in pixel (0, 0) e.g. the top-left. Because TGB scans down the leftmost column looking for non-color-key pixels, you should place the left of your frames against the left of your image. Also, TGB will totally ignore the first row of the image. When TGB encounters a frame, it scans rightwards capturing frames in-between non-color-key pixels. For this reason, you should align the top of all your frames. Frames can be any size but all frames must be no higher than the first frame extracted from that row. When TGB is scanning either down the first columns looking for rows to scan or actually scanning rows, it doesn't matter how many color-keys are between frames.

Phew! You may need a little time to absorb that but it shouldn't be that bad after you've read it a few times. As a summary, here are the rules to follow when creating *"key*" images:

- First, it should be obvious that you need to put your color-key in pixel (0, 0) e.g. the top-left.
- Because TGB scans down the leftmost column looking for non-color-key pixels, you should place the left of your frames against the left of your image.
- TGB will totally ignore the first row of the image.
- When TGB encounters a frame, it scans rightwards capturing frames in-between non-color-key pixels. For this reason, you should align the top of all your frames against the first one in that row.
- Frames can be any size but the next row of frames must be below the tallest frame in the previous row otherwise part of the tall frame will be extracted as part of the next row.
- When TGB is scanning either down the first columns looking for rows to scan or actually scanning rows, it doesn't matter how many color-keys are between frames.
- Although not essential, it's a good idea to "fill" non-frame space with your color-key.

To understand these rules and their implications, read through them again looking at the example image below as a guide.



Give yourself a pat on the back; you've now completely covered the *"key"* image-mode!

"LINK" Image-Mode

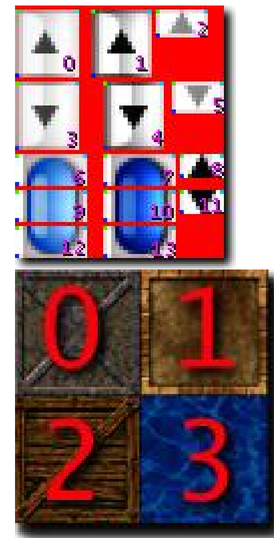
With our review of TGB image-modes, something very important becomes evident. It looks like an image-map is restricted to a single incoming bitmap from which frames are extracted. If you thought this then you'd be correct but you'd also be wrong! In-fact, TGB does limit image-modes to acquire from a single incoming image but it also lets you do something extremely flexible, it allows you to link-together image-maps!

Now at first, this may not seem that important but think about it; TGB allows you to gather-up your frames using various techniques to extract the frames and then allows you to refer to all those frames using a single object that you're already familiar with; the imageMap!

Okay, let's look at how this works; let's start by assuming we've got a couple of image-maps already, inventively called "myImageMap1" and "myImageMap2"; these look like this:

```
datablock t2dImageMapDatablock( myImageMap2 )
{
   imageName = "t2d/data/images/guiMap"
   imageMode = key;
};
```

```
datablock t2dImageMapDatablock( myImageMap1 )
{
   imageName = "t2d/data/images/tileMap"
   imageMode = cell;
   cellWidth = 128;
   cellHeight = 128;
   cellCountX = 2;
   cellCountY = 2;
};
```



As you can see, these image-maps different image-modes of *"key"* and *"cell"* to acquire their frames but the good news is that a linked image-map doesn't care at all! It's important to note that in this example, we're only using two image-maps but you can use as many as you like, the minimum being two.

The first image-map contains 14 frames from "0" to "13" whilst the second contains 4 frames from "0" to "3". Now we've configured these, let's define an image-map to link them up. Here's how we'd do it:

```
datablock t2dImageMapDatablock( myLinkedImageMap )
{
   imageMode = link;
   linkImageMaps = "myImageMap1 myImageMap2";
};
```

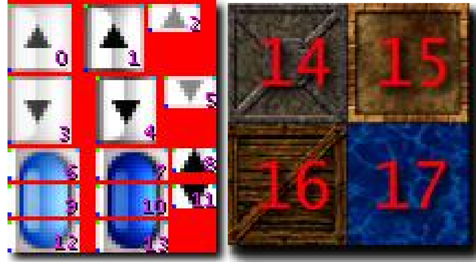
As you can see, we selected the image-mode *"link"* and then specified the image-maps we wanted to link together as a string with space-separators. Note that we didn't specify the *"imageName"* field even though we started by saying that this isn't an optional field! Okay, we lied a little, so let's clarify; *"imageName"* is mandatory for all image-modes except *"link"* mode as it's simply not needed here.

Something that's extremely important to understand is that the referenced image-maps **must** exist when the link image-map is configured, in other words, define a linked image-map after you define the image-maps that it links together in the *"linkImageMaps"* field.

Now believe it or not, that's all we need to do! You can use this image-map exactly as you would any other image-map with no other configuration needed. You can still refer to the linked image-maps "myImageMap1" and "myImageMap2" as normal if you wish; they've not been altered at all.

Now that we know how to setup a linked image-map, let's talk about how the frame numbers are organized as it's these we need to access. Quite simply, each linked image-map specified appends its frame numbers to the frame list of the link image-map.

This sounds a little confusing so using our example above, here are the linked image-maps with the frame numbers used in our new "myLinkedImageMap" image-map:



As you can see, the first set of frames are "0" to "13" which correspond to the frames from the image-map we first specified e.g. "myImageMap1". Now comes the interesting part which is the fact that the frames "0" to "3" from the "myImageMap2" are now referenced by using frames "14" to "17". As you can see, these have been appended to the frames list.

Let's go into a little more detail on exactly what happened here. The first thing to understand is that linking image-maps in this way doesn't duplicate the image-map images either in memory or in your graphics card. Linking image-maps is very efficient and uses relatively small amounts of memory. The next point is that the original image-maps are not altered in anyway, in-fact, you can still access them as normal. You should understand that accessing frame "0" from "myImageMap1" is the same as accessing frame "14" from "myLinkedImageMap". Yeah? If not, you might want to go back and read this section again as this is an important concept.

A cautionary note now is that TGB will not pack the frames from multiple image-maps together to reduce any wasted space. If it were to do this, it would be duplicating the imagery and therefore doubling the memory consumption both on your computer and your graphics card. The caution here relates to the fact that the original textures from the linked image-maps are used when selecting frames. This means that you need to be careful that you're not producing lots of texture-swaps because you're constantly swapping between textures. Typically, this problem only becomes a problem if you're really pushing the limit as to how many frames you're rendering per screen-update so initially, it's probably not something you want to be worrying about.

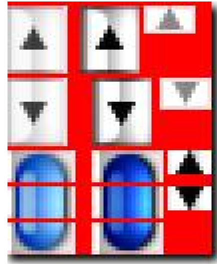
Now that you know how to link-up different image-maps, you should feel confident that you can have different images for your player walking left and right and perhaps jumping and link them all together into a single image-map that your player sprite can refer to.

Happy linking!

"frameCount"

Now that you've seen the various image-modes available and the way to organize your images for frame extraction, the more cautious of you may be asking, "How do I verify I've got all my frames?"

Well, this leads us nicely into our next optional field, *"frameCount"*. Let's use the example image in the previous *"key"* image-mode tutorial although this field is relevant to all image-modes. In that tutorial, we had an image with 14 frames like so:



Let's suppose we were to incorrectly setup either the image-map datablock or the image itself which caused us to get something like 13 frames instead of our expected 14 frames. If this were to happen, normally TGB wouldn't do anything because as far as it knows, you've got what you expected. Sometimes this can be hard to track down, especially if the image-map contains lots of frames. You'd only notice if you didn't get the expected frames or your animations looked screwy!

To provide a little sanity and immediate feedback during development, TGB provides the *"frameCount*" field which allows you to specify how many frames you *expect* from your image-map.

- `"frameCount" - Optional; specifies the expected frame-count.`

This hint allows TGB to take immediate action if the expected number of frames isn't found. If this happens, TGB will abort the image-map creation and issue a warning. You'll also then find that objects attempting to use this image-map will also issue warnings tell you that the image-map doesn't exist. A quick scan of the log will show you the offending image-map. You can then take immediate action to rectify the problem.

Finally, here's how we'd used the field:

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/guiMap"
   imageMode = key;
   frameCount = 14;
};
```

So this is a simple field but can save lots of time tracing problems in image-maps.

****** NOTE ******

You can use "frameCount" in "link" mode in which case it refers to all the frames from all linked image-maps!

“filterMode”

This field is slightly different than the ones we've so far discussed. Unlike the others, this field can be changed dynamically at runtime. TGB also provides a nice function call to do the same but we'll get to that later in this document. More importantly, this document is a rendering option, not a frame extraction option, hence you being able to change it dynamically.

There are two types of filter-modes that you can use, these are:

- `NONE – No Image Filtering.`
- `SMOOTH – Bilinear Filtering.`

It is beyond the scope of this document to detail GPU filtering modes so we'll simply focus on the result of selecting a specific filtering mode.

In "none" mode, the GPU does no filtering. The effects of this don’t become apparent if each pixel from your frame gets rendered to a pixel on the screen. You will start seeing this if you either enlarge/shrink the image on the screen; you'll get an effect called pixilation as shown below:

Although this effect looks poor, sometimes it can be useful artistically in providing an "old school" look. It can also be used to remove certain rendering artifacts that we'll discuss shortly.

Now let's look at "smooth" mode. In this mode, the GPU interpolates between neighbor pixels when rendering which removes the pixilation effect seen above. Also, there's no need to worry about performance here as most if not all graphics cards provide this feature for free e.g. it's as fast as turning off filtering.

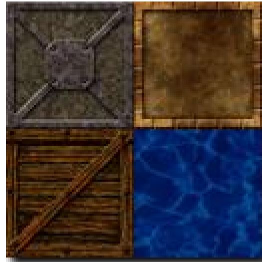
Here's an example of "smooth" filtering in action:

As you can see, the "smooth" filtering removes the pixilation effect but unfortunately tends to blur the images, especially when scaled-up by large amounts.

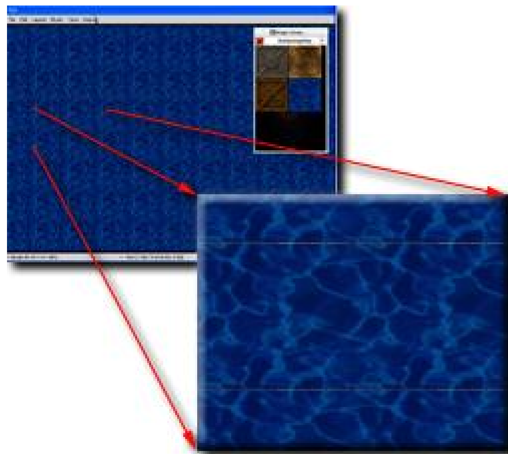
****** NOTE ******
You can use “filterMode” in “link” mode in which case it refers to all linked image-maps!

"filterPad"

Now that you've seen the filter-modes, it's time to introduce a potential problem with using the "smooth" filter-mode. Due to the way that the GPU calculates the "smooth" (bilinear) filter, it has a tendency to sample neighbor pixels. What does this mean? Well, let's look at the image below:



Let's assume we've extracted these four frames and we're using the "water" frame at the bottom-right in a tile-map object and that we're using the "smooth" filter-mode. Well, everything will work perfectly most of the time but occasionally, you'll notice strange lines appearing like so:



What the heck is this? Well, this is an unfortunate artifact created by having multiple-frames not only in the same texture but actually located next to each other. If you look at the horizontal lines in the image above, you may have noticed that they appear to be a brownish color? The reason for this is actually quite simple; when using the "smooth" filter-mode, your graphics card renders each pixel of your frame by looking at the respective pixels' neighbors so that it can smooth what it renders. This works well until the graphics card tries to render pixels at the edge of your frame where the neighbor pixels are actually part of another frame!

So we started this document by saying that TGB will try to pack all these frames together inside your graphics card but now we're saying that this causes problems! What's going on here? Well, packing frames together into as few textures as possible is far more important than the artifacts produced by doing so but with that said, we can't live with these artifacts so we need a way to stop them.

Thankfully, it's relatively simple to fix this problem; if you surround each frame with a copy of the pixels on the border of the frame, when the filter looks at the edge of the frame and therefore looks at the neighbor pixels to calculate the "smooth" version, these neighbor pixels are identical as they are just a copy of the frames border which means, you get no nasty artifacts!

Torque Game Builder – Image Maps

This sounds great, how do I get this to work then? Well, as you may have guessed, this section is entitled "filterPad" and it's this flag that allows you to instruct TGB to achieve "padding" of border pixels to remove the artifact like so:-

```
datablock t2dImageMapDatablock( myImageMap )
{
   imageName = "t2d/data/images/tileMap"
   imageMode = cell;
   cellWidth = 128;
   cellHeight = 128;
   filterPad = true;
};
```

Note that filterPad current defaults to false (off) but this default may change in the future. To ensure you get the same effect independent of any future updates, you can explicitly specify the "filterPad" but note that it's not highlighted here meaning it's not a mandatory field.

You may be wondering if there are any side-effects of using this option and you'd be correct for asking this question. There are indeed some side-effects, although not visual, of using doing this. We'll discuss the frame-packing process later in the document but it should be evident that these border pixels take a little more room in the final textures although only a relatively small amount.

The greater potential trouble is related to the previous "technical note" you may have read. In this note, we mentioned that a majority of graphics card require the textures on your graphics card to be a power-of-two in size e.g. 2,4,8,16,32,64,128,256,1024,2048,4096 etc. Note that we're talking about the texture that TGB uploads to your graphics card and **NOT** the bitmap you specify in the image-map using "imageName"!

If this condition is to be met, TGB needs to try to pack your frames into textures that are appropriately sized which isn't a problem for TGB but can unavoidably result in wasted space in the uploaded textures.

As an example of why this happens, imagine that you have the image above with four frames, each of 128 wide by 128 high and they all fit neatly into a 256 wide by 256 high image. Without frame-padding, this resultant texture will pretty much be the same but with frame-padding on, each frame has an additional single-pixel border surrounding it resulting in frames that are 130 wide by 130 high in size.

Simply putting frames into the same arrangement as the original image would result in an image that is (130x2) x (130x2) = 260x260 in size. This is slightly bigger than the previous power-of-two size of 256 so the next feasible power-of-two size is 512. Fitting of a 260x260 set of frames within a 512x512 image results in (512^2 x 260^2) = 194544 wasted pixels.

****** NOTE ******

We keep referring to "pixels" when we talk about the texture but in-fact, this terminology is incorrect. A "pixel" is a "PIctureELement" e.g. something that is on-screen. The correct terminology to use is "texel" which is short for "TEXture Element". The reason we separate the two is that it's perfectly normal to render a single "texel" that appears across multiple "pixels". This can be confusing if you've not seen this before and therefore we'll keep using "pixel" to mean "texel". For those who understand this, please forgive the misuse!

All these numbers are hard to visualize so let's show this another way:

As you can see, fitting these four 128x128 frames that are padded to 130x130 results in a required area of 260x260 which doesn't fit in a 256x256 so we need to use a 512x512 therefore wasting space as shown by the checker-pattern. Of course, TGB does a much better job than just arranging the frames as shown! In-fact, T2Ds intelligent packing routine packs these frames with only 129008 wasted pixels which is 65536 less than the above example.

"preferPerf"

Although TGB will pack your frames totally automatically, it does provide a field that allows you to give it a hint as to whether you prefer the image-map to have higher performance or least-waste. With the *"preferPerf"* set to *true* (on), it tells TGB that you prefer performance. With the flag set to *false* (off), it tells TGB that you prefer least-waste.

In "performance" mode, TGB will make the best attempt to minimize the number of textures used which reduces the need to swap between textures when rendering the frames contained within it. It is important to note that in "performance" mode, any performance increase may be negligible but can be potentially much higher for frames used in objects such as tile-maps.

In "least waste" mode, TGB will make the best attempt to minimize any wasted (unused) pixels in textures. Although TGB will attempt to keep the number of textures required as low as possible, it will ultimately prefer the least waste.

"preload"

Although you can happily leave TGB to deal with the memory used by image-maps, there comes a time when you might want to provide some hint on how it's handled. By default, when you create an image-map, TGB compiles all the imagery and packs it into textures which are then activated so you can start using it immediately. The trouble is that when the textures are activated, they occupy memory. If you define all your image-maps at the start of your game, they can occupy considerable memory. This is essentially *preloading* in action. The *"preload"* field allows you to control this aspect; if you set this field to false (off) then TGB will compile and pack the image-maps but it won't load the textures.

So when will it? TGB will load the textures as soon as any object references the texture; if you were to assign a frame to a sprite for instance. The downside here is that it can take a little time to activate the textures, dependent upon the current machines performance. The good thing though is that you use as little memory as possible right up to the point where you use it. Of course, this is totally your choice and should be used with care as **not** preloading things can potentially hurt performance if TGB ends up doing lots of work activating textures when, as far as the user is concerned, it should be creating sprites at a critical moment.

You can use "$prefs::T2D::imageMapPreloadDefault" to control the default value for the "preload" field.

"allowunload"

This field is closely linked with *"preload"*. As mentioned above, when **not** preloading textures, textures will be activated when a texture-frame is referenced by a TGB object. What was not discussed was that when the TGB object no longer references the texture-frame, the texture is deactivated meaning it disappears from memory. Don't get this confused with the image-map being destroyed, this isn't the case, simply the texture-data is removed from memory and the image-map is still around. This means that when you don't preload textures, they get automatically activated/deactivated as sprites reference them. You can use "allowUnload" to control whether textures are deactivated when an object no longer references it.

You can use "$prefs::T2D::imageMapAllowUnloadDefault" to control the default value for the "allowunload" field.

Memory Primer

Now that we've discussed the fields “preload” and “allowUnload”, let's just clarify exactly how to use them. When you're interested in how much memory your image-maps use at any one time then these fields become very valuable.

Let's assume you want your image-map to be activated immediately when you create it. Your image-map would look like this...

```
datablock t2dImageMapDatablock( myImageMap )
{
   ...
   ...
   preload = true;
};
```

Now let's assume you want your image-map to only activate when used and deactivate when not used. Your image-map would look like this...

```
datablock t2dImageMapDatablock( myImageMap )
{
   ...
   ...
   preload = false;
   allowUnload = true;
};
```

Now let's assume you want your image-map to only activate when used but stay activated when not used. Your image-map would look like this...

```
datablock t2dImageMapDatablock( myImageMap )
{
   ...
   ...
   preload = false;
   allowUnload = false;
};
```

So what do we mean when we say the image-map is “referenced” or “used”? Well, it's quite simple. If you assign the image-map to a t2dStaticSprite, t2dScroller or indeed, any TGB object, the image-map will recognize this and activate (assuming you're not using “preload”). Now if you change to another image-map or you destroy the TGB object that references the image-map, it will be deactivated (assuming you're not using “preload” and using “allowUnload”). It's that simple!

So what happens when multiple objects use the same image-map? Well, TGB counts the references that objects use. The first one can activate the texture so that's easy but what about deactivating when multiple objects have a reference? Again, it's pretty simple; when an object releases the reference to the image-map, TGB will check to see if the image-map has any references. If it does then TGB does nothing but if there are no references then TGB can deactivate the texture. Of course, as specified, this is dependent upon the options “preload” and “allowUnload”.

So as you can see, TGB provides a nice and simple mechanism to specify when image-map textures are activated/deactivated. You can specify this per image-map or globally using preferences.

Advanced Development

Okay, now that we've covered the image-map fields, it's easy to think that you've got everything you need and to some degree, you can take what you've learned so far and stop here. Please feel free to do that as we've definitely covered all you'll need to get started and enjoy TGB. When you're ready, come back and we'll discuss ways of getting more advanced information/control over your image-maps.

Welcome back!

For here on we'll be discussing the more advanced features of image-maps that can provide vital information when configuring image-maps, not only for debugging but to provide an interface to what's going on behind the scenes. This kind of information can enable you to write various tools to help you configure your image-maps.

Let's start with a discussion of what we mean when we say that TGB "packs" your frames into textures for the graphics card. When TGB looks at your image-map datablock, it finds some vital information. The first is the image-name which defines the bitmap that contains your frames. It then looks at the image-mode which tells TGB how to locate your frames within the specified bitmap. It then goes away and compiles a list of frames and their location in the specified bitmap.

Now comes the complex part! TGB now takes into account your preference of either performance or least-waste as well as whether your frames need padding and starts the process of "packing" your frames into textures. At this point TGB interrogates the maximum sized texture that your hardware can support. If you want to do this yourself, TGB provides the following function (note that you can find more information on this call in the TGB reference documentation):

```
%maxSize = getT2DMaxTextureSize();
```

Now that TGB knows the maximum-sized textures that it can create, it iterates through various combinations of texture sizes noting either how much waste there is or how few textures are created according to the preference rules. Iterating lots of frames through all the variations of texture sizes can be computationally expensive so TGB has some smarter algorithms that enable it to reduce the potential fits available.

When TGB has found the best solution, it starts the process of copying the frames from your original image into the appropriate locations in the new texture(s). It also adds any frame-padding if appropriate. When this is complete, TGB uploads these textures to your graphics card and compiles an internal list so that it can easily refer to the correct texture by frame number. TGB has then completed everything needed to compile an image-map.

During this packing process, TGB can encounter several problems. If a problem occurs, TGB will abort the image-map compilation and remove any resources that it current holds. The result of this will be an error in the console such as:

```
"Register object failed for object myImageMap of class t2dImageMapDatablock."
```

When you see this, you know that the image-map named "myImageMap" failed. What you'll probably see then are cascaded failures when you've subsequently tried to use the image-map in other TGB objects but the important one here is the initial failure. TGB is based upon the T3D engine which provides the message as shown above. Now although this message is useful in knowing that a problem occurred, it doesn't say what the problem was.

Luckily, TGB provides more information on any problems encountered. It does this in several ways. The first is that TGB issues script-callbacks which allow you to directly control what happens when any of your image-maps fail at runtime. Secondly, you can instruct TGB to additionally echo any error to the console.

Let's start with the script-callbacks; there are three callbacks in total, these are:

- `onImageMapPackStart() - Packing has started.`
- `onImageMapPackEnd() - Packing has ended.`
- `onImageMapError() - A Packing error occurred.`

These callbacks enable you to monitor the packing start and stop of each image-map as well as any errors that may occur. Due to the fact that image-maps are not valid objects until they successfully complete the packing process, these callbacks are not passed the typical "%this" implicit object identifier. The callbacks appear in a specific order of "*onImageMapPackStart"*, *"onImageMapPackEnd"* and *"onImageMapError"*. They are used like so:

```
// Start Packing.
function onImageMapPackStart( %imageMapName )
{
   echo( "Packing has started for image-map" SPC %imageMapName );
}

// End Packing.
function onImageMapPackEnd( %imageMapName )
{
   echo( "Packing has ended for image-map" SPC %imageMapName );
}

// Packing Error.
function onImageMapError( %imageMapName, %errorNumber, %errorDescription )
{
   echo( "An error occurred in image-map" SPC %imageMapName );
   echo( "The error number was" SPC %errorNumber );
   echo( "The error description was" SPC %errorDescription );
}
```

The start/end packing calls are pretty simplistic and simply allow you to monitor which image-maps are being processed and when they complete. The end-packing call will happen irrelevant of any error occurring. Following the end-packing call, you may get an error callback. Let's go into a little more detail on this error callback.

When an image-map error occurs, it immediately aborts and results in the packing process ending and an error callback. Because image-maps stop on the first error they encounter, you'll only get a single error callback. Of course, if you resolve the problem and try again, you may then get another error.

All of the image-map callbacks return the name of the image-map as the first parameter to the callback which is useful in identifying which image-map has failed. As mentioned above, the image-map doesn't yet exist as an object to the scripts so you cannot refer to it directly so the name is your only reference.

The error callback provides two additional parameters which are extremely valuable in determining what went wrong. The first is the "error number" which is an integer starting at 1 which uniquely identifies each error. The second is a textual description of the error itself. You may want to provide specific responses to certain errors at runtime so the "error number" is more useful in this case.

Here's a list of potential error numbers and their respective description:

```
1.  - "Invalid Mode; must be FULL, KEY, CELL or LINK!"
2.  - "Invalid Bitmap File!"
3.  - "Frame Width Too Big; Unsupported by current hardware!"
4.  - "Frame Height Too Big; Unsupported by current hardware!"
5.  - "Filter-Padded Frame Width Too Big; Unsupported by current hardware!"
6.  - "Filter-Padded Frame Height Too Big; Unsupported by current hardware!"
7.  - "Frame Width Too Big; Larger than selected limit!"
8.  - "Frame Height Too Big; Larger than selected limit!"
9.  - "Filter-Padded Frame Width Too Big; Larger than selected limit!"
10. - "Filter-Padded Frame Height Too Big; Larger than selected limit!"
11. - "Reported Maximum Hardware Texture Size is invalid!"
12. - "Selected Maximum Texture-Size is larger than the current hardware limit!"
13. - "Invalid CELL Offset X!"
14. - "Invalid CELL Offset Y!"
15. - "Invalid CELL Dimension!"
16. - "Invalid CELL Quantity!"
17. - "Invalid CELL Capture!"
18. - "Failed to get separator color!"
19. - "No frames have been acquired!"
20. - "Number of frames acquired differs from that specified!"
```

As we mentioned earlier, you can also ask TGB to additionally echo these errors to the console. Perhaps you're not interested in using the script callbacks and simply want to monitor your console log. TGB provides several preferences for image-maps, one of these being the echoing of image-map errors. Let's introduce these preferences now:

- `$prefs::T2D::imageMapEchoErrors`
- `$prefs::T2D::imageMapShowPacking`
- `$prefs::T2D::imageMapDumpTextures`
- `$prefs::T2D::imageMapFixedMaxTextureSize`
- `$prefs::T2D::imageMapFixedMaxTextureError`
- `$pref::T2D::imageMapPreloadDefault`
- `$pref::T2D::imageMapAllowUnloadDefault`

Hopefully, you'll already know about your preference file(s). If you don't then you simply need to know that it is a script file where TGB exports any variables from the "$prefs::" namespace when you exit TGB. You can specify variable parameters to the TGB engine here therefore image-maps expose the previous five preferences here.

You can more detail on these preferences in the TGB reference documentation, nevertheless let's go through each of these preferences:

"$prefs::T2D::imageMapEchoErrors"
This preference tells TGB to echo any error to the console. TGB will still produce an error callback.

"$prefs::T2D::imageMapDumpShowPacking"
This preference tells TGB to echo image-map statistics at the start and stop of packing. These statistics contain comprehensive information on the packing process and can be invaluable in understanding what TGB is doing with your frames. As well as frame locations, texture-page dimensions and counts, TGB also provides timings for the packing process.

"$prefs::T2D::imageMapDumpTextures"
This preference tells TGB to dump the resultant textures that contain the packed frames to disk before uploading to your graphics card. You can use these images to visualize what exactly TGB is doing with your frames.

TGB will always dump these images to the root script directory into a subdirectory called "imageMapDump". The output images will be the same format as the original input image.

The nomenclature for these files is:

<image-map name>_Page_<Texture Page>.<original image extension>

An image-map called "myImageMap" that produces two textures from frames originally contained in a ".png" file will output the following files:

"myImageMap_Page_0.png"
"myImageMap_Page_1.png"

Note that TGB does not clear this directory prior to outputting files therefore files from previous executions may still be contained. Any files of identical names will be overwritten without confirmation.

"$prefs::T2D::imageMapFixedMaxTextureSize"
This preference tells TGB to override the maximum supported hardware texture size (MSHTS) in favor of this specified size. This value must be equal or less than the MSHTS. Additionally, this value **must** be a power-of-two. If it isn't, TGB will automatically raise it to the next power-of-two. It defaults to zero which tells TGB to ignore it and to use the MSHTS instead.

Although this preference would probably be left at its default most of the time, it does allow you to override under certain conditions. For instance, you may find that performance of the MSHTS is not adequate on specific hardware. You can use the scripts to identify this hardware and adjust this preference accordingly.

"$prefs::T2D::imageMapFixedMaxTextureError"
This preference is directly related to the previous preference. It allows you to override the situation where the specified maximum texture size is actually higher than the MSHTS. In this situation, TGB cannot honor the request and must do something to resolve the problem. TGB has two choices; defer to the MSHTS or abort with an error. This preference controls which option TGB chooses. The default is *false* (off) which means TGB will defer to the MSHTS. When set to *true* (on), TGB will abort with an error.

"$prefs::T2D::imageMapPreloadDefault"
If you don't specify the “preload” field then this preference allows you to select the value used by default. You can essentially use this setting to select a global default for the “preload” field, overriding it where necessary using the “preload” field.

"$prefs::T2D::imageMapAllowUnloadDefault"
If you don't specify the “allowUnload” field then this preference allows you to select the value used by default. You can essentially use this setting to select a global default for the “allowUnload” field, overriding it where necessary using the “allowUnload” field.

Now that we've covered both callbacks and preferences that can be used to monitor the image-map packing process as well as any errors generated, let's finish by discussing the functions that image-maps provide to allow you to control and interrogate them.

Here's a complete list of the functions provided by image-maps:

- `setFilterMode() – Set GPU Filter Mode.`
- `getFilterMode() – Get GPU Filter Mode.`
- `getImageMapMode() – Get Image-Map Image-Mode.`
- `getSrcBitmapName() – Get Source Image Bitmap Name.`
- `getSrcBitmapSize() – Get Source Image Bitmap Size.`
- `getFrameCount() – Get Total Frame Count.`
- `getFrameSize() – Get Specific Frame Size.`
- `getTexturePageCount() – Get Total Texture-Page Count.`
- `getTexturePageSize() – Get Specific Texture-Page Size.`
- `getTexturePageFrameCount() – Get Frame Count on specific texture-page.`
- `getFrameTexturePage() – Get Texture-Page that specific frame resides.`

You can find more information on these functions in the TGB reference documentation, nevertheless let's go through these functions individually:

"setFilterMode(%mode) / getFilterMode()"
These two functions allow you to set/get the filter mode to either "none" or "smooth". This replicates the image-map field *"filterMode"* such that you can use either the field or the functions to set/get the filter-mode.

"getImageMapMode()"
This function retrieves the image-map mode. This replicates the image-map field *"imageMode"* such that you can use either the field or the function to get the image-map mode. Note that changing the field after the image-map has been created has no effect whatsoever.

"getSrcBitmapName()"
This function retrieves the name of the source image bitmap used to extract frames. This replicates the image-map field *"imageName"* such that you can use either the field or the function to get the image-name. Note that changing the field after the image-map has been created has no effect whatsoever.

"getSrcBitmapSize()"
This function retrieves the size of the source image bitmap used to extract frames. An 800 wide x 600 high bitmap would be returned as "800 600".

"getFrameCount()"
This function retrieves the total frame count calculated by the image-map.

"getFrameSize(%frame)"
This function retrieves the size of the specified frame. The frame-number is zero-based and is valid to the value returned by *"getFrameCount()"* minus one.

"getTexturePageCount()"
This function retrieves the total texture-page count calculated by the image-map.

"getTexturePageSize(%page)"
This function retrieves the size of the specified texture-page. The page-number is zero-based and is valid to the value returned by *"getTexturePageCount()"* minus one.

"getTexturePageFrameCount(%page)"
This function retrieves the total frame count on the specified texture-page. The page-number is zero-based and is valid to the value returned by *"getTexturePageCount()"* minus one.

"getFrameTexturePage(%frame)"
This function retrieves the index of the texture-page where the specified frame resides. The frame-number is zero-based and is valid to the value returned by *"getFrameCount()"* minus one. The page-number returned is zero-based and is valid to the value returned by *"getTexturePageCount()"* minus one.

It may not be obvious how you'd call image-map datablock functions but being as they are fundamentally an object within TGB, they can be used in much the same way as you would use any other object in TGB. As an example of this calling process, here's how you'd set the filter mode on an image-map:

```
myImageMap.SetFilterMode( smooth );
```

As you can see, there's a wealth of information that can be retrieved from image-maps but it is important to understand that if any error occurs during the image-map creation, the image-map object will not be available therefore neither will these functions.

Conclusion

We've now come to the end of our discussion on TGB image-maps and we hope that we've provided enough information for TGB novices and veterans alike. TGB image-maps are the "bread and butter" of using TGB so a fundamental understanding of how to use them is vital. Because they are so vital, they've become more and more powerful offering more ways to define your imagery and they'll get even more powerful in the future.

Don't forget that you can always find information on TGB image-maps in the TGB reference documentation.

All the best everyone and happy TGB'ing! **J**