**WORK RAM**

The engine can be customized to allocate a variable amount of WRAM, in a configurable number of pools and sizes, to efficiently accommodate the game's dynamically allocated objects.

The maximum WRAM that can be allocated for the MemoryPool, before overflowing the stack, is, in the current engine's version, below the 52 Kb, that is without any object in the game's stage, nor any texture preloaded. Populating the game's stages will require to shrink down the MemoryPool's size to make room for the growing stack.

To modify the memory pools available, and theirs sizes, define the following MACROS in the config.h file of the game, as the example shows:

#define \_\_MEMORY\_POOLS 3

#define \_\_MEMORY\_POOL\_ARRAYS \

\_\_BLOCK\_DEFINITION(256, 64) \

\_\_BLOCK\_DEFINITION(128, 64) \

\_\_BLOCK\_DEFINITION(64, 64) \

#define \_\_SET\_MEMORY\_POOL\_ARRAYS \

\_\_SET\_MEMORY\_POOL\_ARRAY(256) \

\_\_SET\_MEMORY\_POOL\_ARRAY(128) \

\_\_SET\_MEMORY\_POOL\_ARRAY(64) \

#define \_\_MIN\_BLOCK 64

This configuration will create 3 pools, one that holds objects with sizes of 0 up to 64 bytes, other that holds object of sizes between 68 and 128 bytes, and finally, one that holds objects of sizes between132 and 256 bytes. The size of each pools is the product of both numbers passed as arguments to the \_\_BLOCK\_DEFINITION MACRO; in this example that amounts to:

256 \* 64 = 16384

128 \* 64 = 8192

64 \* 64 = 4096

TOTAL = 28672 bytes

**ANIMATION ALLOCATION TYPES**

**CHAR ALLOCATION TYPES**

The engine abstracts Virtual Boy's VIP's memory sections in the following classes:

* CHAR memory: CharSet
* BGMAP memory: BgmapTexture
* OBJ memory: ObjectTexture
* WORLD memory: Sprite (BgmapSprite and ObjectSprite)

To allocate CHAR memory, the CharSetManager's getCharSet method must be called, which, depending on the CharSetDefintion's allocation type, will create a new CharSet or return an existing one.

To allocate BGMAPmemory, the BgmapTextureManager's getTexture method must be called, which, depending on the CharSetDefintion's allocation type, will create a new Texture or return an existing one.

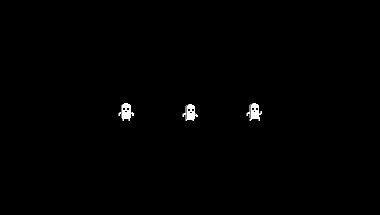
Both CharSet and BgmapTextures are allocated using reference counting in order to reduce both VRAM and WRAM footprint.

Each object segment is allocated by an ObjectSpriteContainer.

\_\_ANIMATED\_SINGLE

When using this animation type, the engine allocates a new CharSet and Texture for each request, and each time a new frame must be show, the engines writes directly to CHAR memory.

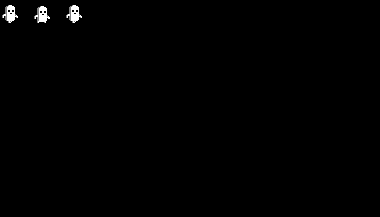
For example, each one of the following AnimatedSprite has its own Texture and CharSet:



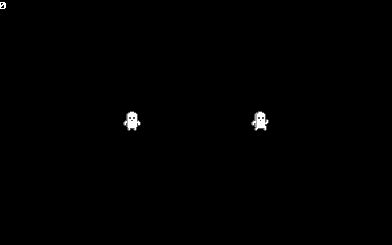
The inspection of the CHAR memory reveals that the CHARs corresponding to the current frame of animation of each AnimatedSprite, have been loaded, even if they belong to the same CHAR definition:



If all AnimatedSprites are BgmapSprites, then, the inspection of BGMAP memory will show that only one frame of animation is loaded for each AnimatedSprite:



If the AnimatedSprites at both ends are ObjectSprites, then, the inspection of OBJ memory will show the appropriate frame of animation for each AnimatedSprite:



Usages:

* This type of animation should be used for AnimatedSprites with too many animation frames or whose graphics occupies too many CHARs.

Limitations:

* Textures that use CharSets with this type of allocation must not be preloaded, since the preloaded instance will be unusable.
* Char definition must not be optimized and each group of CHARs that form a frame of animation must preserve the order of the first frame's CHARs as specified by the BGMAP definition. Supposing that a Sprite has 3 animation frames, the Texture's size is 3x3 CHARS, and the BGMAP definition looks like:

0 1 2

3 4 5

6 7 8

where each number signifies and index in CHAR memory; then, if the 9 CHARs (0-8) that form the first frame of animation have the following appearance:

:frames:f0.png

then, the second group of CHARs (9-17), that form the second frame of animation, must look like:

:frames:f2.png

and finally, the third group of CHARs (9-26), that form the last frame of animation, must look like:

:frames:f4.png

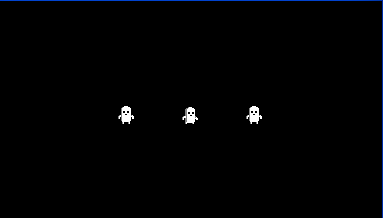
Downsides:

* Performance heavy.

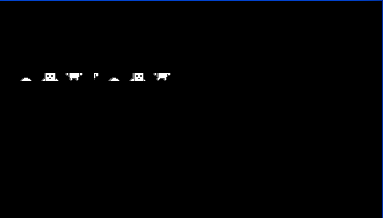
\_\_ANIMATED\_SHARED

When using this animation type, the engine allocates a new CharSet once (and only a Texture if BgmapSprites are used); and for each new request with the same char definition, returns the same reference(s). Each time a new frame must be show, the engines writes directly to CHAR memory, and every Sprite that uses the same CharSet wiil display the change.

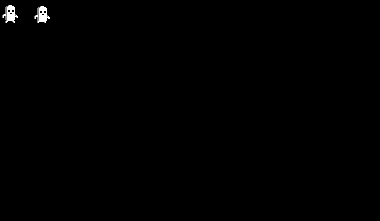
In the following example, the AnimatedSprite in the center uses an \_\_ANIMATED\_SINGLE CharSet, while the AnimatedInGameEntities at both ends use the same \_\_ANIMATED\_SHARED CharSet:



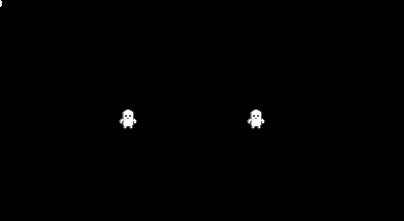
The inspection of the CHAR memory reveals that the CHARs corresponding to the current frame of animation of the first and second AnimatedSprites, have been loaded, being the first CHAR's sequence the one shared by the AnimatedSprites at both ends of the screen:



If all AnimatedSprites are BgmapSprites, then, the inspection of BGMAP memory will show that only one BgmapTexture has been loaded for the AnimatedSprites at both ends of the screen:



If the AnimatedSprites at both ends are ObjectSprites, then, the inspection of OBJ memory will show the same frame of animation for both:



Usages:

* This type of animation should be used when there are many instances of the same AnimatedSprite definition, which has with too many animation frames or whose graphics occupies too many CHARs, and whose animations can be synchronized.

Limitations:

* Playing an animation in one AnimatedSprite instance will affect the others, and playing animations in different instances will waste processor time, since only the last rendered AnimatedSprite's current animation frame will be shown.

Downside:

* Performance heavy.

Remarks

* If only one AnimatedSprite instance use the CharSet, it behaves exactly as the \_\_ANIMATED\_SINGLE type.

\_\_ANIMATED\_SHARED\_COORDINATED

This animation type works exactly the same as the \_\_ANIMATED\_SHARED, but for each CHAR definition, the engine spawns an AnimationCoordinator that ensures that only one AnimationController is playing an animation at any given time.

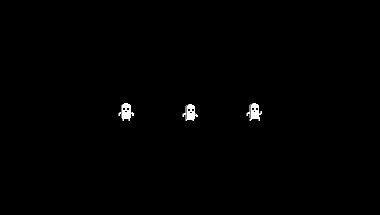
Usages:

* The same as \_\_ANIMATED\_SHARED\_COORDINATED, but saves processor's time when multiple AnimatedSprites must be synchronized.

\_\_ANIMATED\_MULTI

When using this animation type, the engine allocates a new CharSet once (an only Texture when using BgmapSprites); and for each new request, with the same char definition and allocation type, returns the same reference(s). Depending on the Sprite's type, when a new frame of animation must be shown, the engine either modifies the WORLD's mx and my values, or writes to OBJ memory.

In the following example, the AnimatedSprite in the center uses an \_\_ANIMATED\_SHARED CharSet, while the AniamtedSprites at both ends use the same \_\_ANIMATED\_MULTI CharSet:



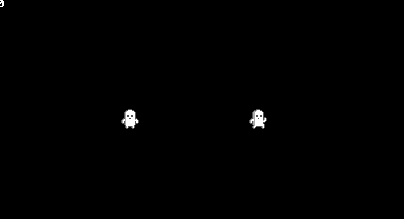
The inspection of the CHAR memory reveals that all the CHARs used by all the animation frames have been loaded into CHAR memory for the AnimatedSprites at both ends of the screen (the last CHARs correspond to the AnimatedSprite at the center, that uses an \_\_ANIMATED\_SINGLE CharSet):



If all the AnimatedSprites are BgmapSprites, then, the inspection of BGMAP memory will show that all frames of animation have been loaded for the AnimatedSprites at both ends of the screen (the last frame corresponds to the AnimatedSprite in the center, that uses an \_\_ANIMATED\_SINGLE CharSet):



If the AnimatedSprites at both ends are ObjectSprites, then, the inspection of OBJ memory will show the corresponding animation frame has been loaded for each of the AnimatedSprites at each end of the screen:



Usages:

1. This type of animation should be used for AnimatedSprites with too many animation frames or whose graphics occupies too many CHARs, but when their instances must not be necessarily synchronized.

Limitations:

* When using a \_\_WRLD\_AFFINE BgmapSprite, each time that an animation frame have to be rendered, the affine table must be computed.

Downsides:

* May consume too much CHAR memory.

**STAGE**

**Adding entities to the stage**

To add entities to the stage, either use the Stage\_addEntity method, or the Entity\_addChildFromDefinition, depending on the desired parent for the new entity.

**Deleting entities**

Since the whole stage is composed of containers, the safe way to delete them is to use the Container\_deleteMyself method, otherwise the container could be being deleted from within a list's loop, causing undefined behavior.

Don't delete directly entities added to the parenting hierarchy by using the \_\_DELETE macro.

**ERROR DIAGNOSIS**

Since the engine makes heavy use of pointer's logic, it is really easy to trigger difficult to find bugs. In order to mitigate this issue, the engine provides the following aids:

**Asserts**

Use the ASSERT macro to check every pointer or variable which can be trouble some, specially, place an ASSERT checking that the "this" pointer passed to the class's methods is valid.

**Initialize everything**

One of the most difficult, and common source of hard to diagnose bugs are uninitialized variables; random crashes or completely strange behavior often are caused by not properly initialized variables. To aid the detection of such mistakes, in the config.h file, define the \_\_MEMORY\_POOL\_CLEAN\_UP macro, this will force the engine to put every memory pool's free block to 0 when the game changes its state, so, if the problem solves by defining such macro, the cause is, most likely, a variable not initialized.

**MemoryPool's size**

Another source of strange behavior and crashes occurs when the stack is overflowed. Try to reduce the memory pool's size by a few KB in order to dismiss this stack's overflow as the source of the issue.

**Cast everything**

Because the engine implements class inheritance by accumulation of attributes' definitions within macros, it is necessary to cast every pointer of any given class to its base class in order to avoid compiler warnings when calling the base class' methods; this exposes the program to hard to identify errors. In order to mitigate this danger, cast every pointer before passing it to the base class' method by following this pattern:

BaseClass\_method(\_\_GET\_CAST(BaseClass, object), ...);

When compiling for release, the macro be replaced by a simple C type cast; while for debug, the Object\_getCast method will be called, returning NULL if the object does not inherit from the given BaseClass, raising an exception in the method (which must check that the "this" pointer isn't NULL).