VBJAENGINE

PROGRAMMER'S GUIDE

Overview

The vbJAEngine has been developed to fill the gap between the developer's game logic and the hardware management required to produce visual and audio results on the Nintendo Virtual Boy. The engine is a sprite based engine, which takes care of all the work necessary to display user defined sprite images on screen, allocating graphical and CPU's memory to hold the definitions of such images.

Features

The engine provides the following features for the programmer to take advantage of:

- Automatic Char memory allocation.
- Automatic BGMap memory allocation.
- Automatic world layer assignment based on the object's Z position.
- Automatic frame rate control.
- Memory Pool to allocate memory dynamically.
- Automatic memory allocation for Param tables (used in Affine and H-Bias modes).
- Easy to use printing functions to facilitate debug.
- Sound reproduction of one BGM and up to two FX sounds simultaneously.
- A generic container.
- Messaging system.
- Generic State Machine.
- Automatic collision detection and notification.
- Object Oriented support through the use of Metaprogramming (C MACROS):
    - Inheritance
    - Polymorphism
    - Encapsulation
- Useful classes to speed up the content creation process:
    - Image
    - Background
    - Character
    - Scroll
- Automatic loading/unloading of objects in/outside the screen.
- 3D stages.
- Simple physics simulation:
    - Accelerated/uniform movement
    - Gravity
    - Friction
- Scaling/rotation effects.
- Clocking system.
- Automatic projection/parallax/scale calculation and rendering.
- Customizable perspective/deep effects on real time.
- Frame based animation system with callback support.
- Generic main game algorithm (game loop).

TODO

- Particle system.
- Polygon based engine.
- Tile based collision detection.

Object Oriented Programming

Since the nature of games is better and more easily represented by the OOP paradigm than a structured one, because the former allows for better code reusage and flexibility, and since the available compiler for the VB only supports C, for this engine a set of C MACROS have been created to simulate some of the most visible features provided by C++:

- Inheritance
- Polymorphism
- Encapsulation

In order to use these features, you must be comfortable using some MACRO calls which will be explained next.

Creating a class

Every class in the engine and the game must inherit from a base class called Object or from another class which inherits from it.

Let's create a Mario class which inherits from the Character class provided with the engine
In the header file Mario.h the following macros must be placed:
This will make Mario class to inherit the virtual methods from the Character class.

```
#define Mario_METHODS                                                   \
Character_METHODS;
```

Next, it is necessary to inherit and/or redefine those method's definitions:

```
#define Mario_SET_VTABLE(ClassName)                                     \
Character_SET_VTABLE(ClassName);                                        \
__VIRTUAL_SET(ClassName, Mario, die);
```

This tells the engine that Character has a virtual method called die (Character_die indeed) and we want to redefine that method with our own version of die, thus allowing Polymorphism.

Then you must declare the class with the following line:

```
// A Mario!
__CLASS(Mario);
```

This will define a pointer Mario to a struct, this way the Mario class's implementation is hidden from client code, making it impossible to access private members, which provides Encapsulation.

Then you must declare Mario class's specific attributes, to do so, declare the following MACRO:

```
#define Mario_ATTRIBUTES                                                \
                                                                        \
/* it is derivated from */                                              \
Character_ATTRIBUTES                                                    \
                                                                        \
/* mario has energy */                                                  \
u8 energy;                                                              \
                                                                        \
/* time when an event ocurred*/                                         \
u32 actionTime;                                                         \
                                                                        \
/* hold object */                                                       \
Character holdObject;
```

Notice that all these macros have a ,,\" at the end of each line, you must be careful and always be sure that there are not blank or tab spaces after them.
The reason to declare the class's attributes this way is because this allows to inherit methods/attributes, and at the same time allows to make the attributes private.
Notice that in order for Mario to inherit Character's attributes it includes in Mario_ATTRIBUTES the Character_ATTRIBUTES.

The last thing to be done in the header file is to declare the next methods:

- Allocator
- Constructor
- Destructor

Allocator: all classes must follow the following format (the arguments are optional).

Mario Mario_new(CharacterDefinition* animatedEntityDefinition, int ID);

Constructor: the first argument is mandatory.

void Mario_constructor(Mario this, CharacterDefinition* definition, int ID);

Destructor: the argument is mandatory and must only be one in all cases.

void Mario_destructor(Mario this);

Now it is time to define the class. In the source file Mario.c do as follows:

Include the header file which holds the class's declaration:

```
#include "Mario.h"
```

Define the class:

```
// A Mario! Which inherits from Character
__CLASS_DEFINITION(Mario, Character);
```

Define the allocator:

```
// always call these to macros next to each other
__CLASS_NEW_DEFINITION(Mario, __PARAMETERS(CharacterDefinition*
characterDefinition, int ID))
__CLASS_NEW_END(Mario, __ARGUMENTS(this, characterDefinition, ID));
```

Define the constructor: must always call the parent class's constructor to properly initialize the object.

```
// class's constructor
void Mario_constructor(Mario this, CharacterDefinition*
characterDefinition, int ID){

      __CONSTRUCT_BASE(Character, __ARGUMENTS(this,
characterDefinition, ID));

      this->energy = 1;
      ...
}
```

Define the destructor: must always destroy the parent class at the end of the method.

```
// class's destructor
void Mario_destructor(Mario this){
      // free space allocated here

      // delete the super object
      __DEALLOCATE(Character);
}
```

Virtual calls

The purpose of having OOP features is to allow generic programming through the use of virtual calls to class methods through a base class pointer, for example, the Stage has a list of Entities (from which Character, Background and Images inherit) and it must be able to call the proper update and render methods on those classes, to do so, there are two forms, using a switch statement to determine which object's type being pointer by the parent class pointer (which can means extra info on each object to hold the type, and can be quite cumbersome if we extend to have more kind of Entities). The other way is to use virtual calls to virtual methods:

```
// update each entity's internal state
void Stage_update(Stage this){

	VirtualNode node = VirtualList_begin(this->entities);

	// update each entity
	for(; node ; node = VirtualNode_getNext(node)){

		// update each entity
		__VIRTUAL_CALL(void, Entity,
update,(Entity)VirtualNode_getData(node));
	}
}
```

As you can see, there is only one call to the method, and the one being called depends upon the object's type being currently processed.

Adding objects to the Stage

To add objects to the Stage call the following method:

```
// create an entity in the game's and load it into the Stage
Mario mario =
Stage_addEntity(Game_getStage(Game_getInstance(),marioDefinition,
position, -1);
```

Notice the -1, that is the Entity's ID, the Stage will assign valid IDs to Entities loaded from a GameWorld definition, when you want to create a dynamic Entity (i.e: a Mario's fireball) provide the -1 index.
This method will do the necessary calls to allocate the object in CPU and graphic memory, add it to the stage and assign a world layer to be displayed.

Removing objects from the Stage

Let say our Mario character has been killed and must now to be removed from the Stage, to do so call the following method:

```
// inform the game that I'm dead
Stage_removeEntity(Game_getStage(Game_getInstance()), (Entity)mario,
kDead);
```

Notice the kDead enum, which tells the engine to not load the Entity again if it is within the screen range.

The Game's StateMachine

The Game itself has a StateMachine to hold the actual state of it and execute the proper logic. So after creating the game, and before entering the game's infinite loop, it must enter a valid state:

```
// Main game algorithm
// There should not be need
// to modify this.

int main(void){

	// create the game
	Game game = Game_getInstance();

	// clear sprite memory
	vbClearScreen();
	// turn on the display
	vbDisplayOn();

	// set engine's initial state
	Game_setState(game, (State)PVBCCScreen_getInstance());

	// main game loop
	Game_update (game);

	// end program
	return true;
}
```

As you can see, the game will enter the state PVBCCScreen first, and then will enter the game's main loop Game_update();
PVBCCScreen is a game state which must perform and initialization, execute its logic, and change the game's current state to another one based on user input.
In its initialization stage, PVBCCScreen loads a GameWorld definition which describes the Stage, and the Entities within it:

```
// state's enter
static void PVBCCScreen_enter(PVBCCScreen this, void* owner){

	//clear char and bgmap memory
	vbClearScreen();

	// reset the engine state
	Game_reset(Game_getInstance(), true);

	// load world's entities
	Stage_setup(Game_getStage(Game_getInstance()), &PVBCC_WR);
```

```
	// call the render to setup world entries
	Game_render(Game_getInstance());

	// make a fade in
	Game_FXFadeIn(Game_getInstance(), FADEDELAY);
}
```

The Stage_setup method instructs the stage to load the world defined by PVBCC_WR.

```
StageDefinition PVBCC_WR = {
	// size
	{
			// x
			384,
			// y
			228,
			// z
			10
	},
	//initial screen position
	{
			// x
			0,
			// y
			0,
			//z
			__ZZERO
	},
	//background music
	NULL,

	//entities
	{

		{&PVBCC_LEFT_BG,{SCREEN_CENTER_X, SCREEN_CENTER_Y, 0}},
		{&PVBCC_RIGHT_BG,{SCREEN_CENTER_X, SCREEN_CENTER_Y, 0}},
		{NULL, {0,0,0}},
	},
};
```

A Stage is defined by its size, the initial screen position, a pointer to a BGM, and a list of Entities. There must be always a {NULL, {0,0,0}} at the end of the definition to inform the Stage that the Entity list ends there.

Let's take a look at one of the Entities being loaded.

```
ImageDefinition PVBCC_LEFT_BG = {
	{
		// object's class
		__TYPE(Image),
```

```
                // Sprite
                {
                        // the texture
                        &PVBCC_LEFT_TX,

                        // bgmap mode ( BGMAP, AFFINE, H-BIAS)
                        WRLD_BGMAP,

                        // display mode
                        WRLD_LON,
                },
        },
};
```

The line

__TYPE(Image)

tells the Stage which kind of Entity is currently being loaded so the proper constructor is called.

Then comes the Sprite definition, PVBCC_LEFT_TX is the address of the Texture to use with the Sprite. The Sprite definition dictates the texture display mode (affine, bgmap, h-bias) and the displays on which the image will be shown.

```
TextureDefinition PVBCC_LEFT_TX = {

        // Chargroup
        {
                        // chDefinition,
                        PVBCC_CHARSET_LEFT,

                        // numChars,
                        238,

                        // allocation type
                        __NO_ANIMATED
        },

        // bgmap definition
        PVBCC_MAP_LEFT,

        // cols (max 48)
        48,

        // rows (max 28)
        28,

        //pallet number,
```

```
        0
};
```

PVBCC_CHARSET_LEFT and PVBCC_MAP_LEFT are the names of the const arrays with the char and bgmap definitions produced by programs like VIDE.

Notice that the total cols and rows which represent the image's size, are 8's divisors since each char is 8 pixels wide and 8 pixels height.

Creating Animations

Characters are used to represent Entities which have certain animations and which can move inside the Stage. In order to provide a Character with an animation it must be defined as follow:

```
CharacterDefinition MARIO_MC = {
        {
                {
                        // object's class
                        __TYPE(Mario),

                        // Sprite
                        {
                                // the texture
                                &MARIO_TX,

                                // bgmap mode ( BGMAP, AFFINE, H-BIAS)
                                WRLD_AFFINE,

                                // display mode
                                WRLD_ON,
                        },
                },

                // deep
                5,

                // friction factor
                5.5,

                //collision detection gap
                //up,  down,  left,  right,
                {5,           0,            8,            6,},
                //{0,         1,            0,            0,},

                // in game type
                kMario,
```

```
	},

	// pointer to the animation definition for the character
	&MARIO_ANIM,
};
// a function which defines the frames to play
AnimationFunction MARIO_BLINKING_ANIM = {

	// function's name
	"Blink",
	// number of frames of this animation function
	3,

	// frames to play in animation
	{16, 17, 16},

	// number of cicles a frame of animation is displayed
	5 * __FPS_ANIM_FACTOR,

	// whether to play it in loop or not
	false,

	// method to call function completion
	Mario_blinkDone,
};
// an animation definition
AnimationDescription MARIO_ANIM = {

	// number of animation frames
	28,

	// animation functions
	{
		&MARIO_IDLE_ANIM,
		&MARIO_WALKING_ANIM,
		&MARIO_JUMPING_ANIM,
		&MARIO_FALLING_ANIM,
		&MARIO_SLIDING_ANIM,
		&MARIO_HIT_FRONT_ANIM,
		&MARIO_HIT_BEHIND_ANIM,
		&MARIO_JUMP_BACK_ANIM,
		&MARIO_JUMP_FRONT_ANIM,
		&MARIO_FRONT_ANIM,
		&MARIO_BACK_ANIM,
		&MARIO_BLINKING_ANIM,
		&MARIO_WALKING_HOLD_ANIM,
		&MARIO_BLINKING_HOLD_ANIM,
		&MARIO_IDLE_HOLD_ANIM,
		&MARIO_JUMPING_HOLD_ANIM,
		&MARIO_FALLING_HOLD_ANIM,
		&MARIO_WALKING_FRONT_ANIM,
		&MARIO_WALKING_BACK_ANIM,
```

```
                NULL,
        }

};
```

The AnimationDescription defines how many animation frames comprise the animation, and holds the address of each AnimationFunction. An AnimationFunction must provide a name, the number of frames to be played, the sequence of frames, each frame duration based on the target frame rate, whether the frames must be played in a loop, and a function pointer to be called when the last frame of animation has been displayed.

Animation allocation types

Since there can be Characters with many frames of animations like Mario, and Characters with simple animations like Koopas, there is a need to allocate each kind in different ways.

There are three ways to allocate animations in memory:

__ANIMATED:

Each Character has its dedicated char and bgmap memory. On each animation frame change, the char definition is rewritten with the next. Use this method for character with tons of animations.

__ANIMATED_SHARED:

Both the char and bgmap definitions are shared, and the whole animation frames are allocated in both memories, so in order to display each animation frame, the world layer window is moved to the proper frame in case of bgmap mode being used, or the param table is rewritten in the case of affine mode. This is useful when there is a Character like Koopa which has few chars and few animation frames and there can be lot of the same Character on screen.

__NO_ANIMATED:

Used with static images like backgrounds, logos, etc.

In order to play a specific animation, call the following method:

```
Character_playAnimation((Character)this, "Blink");
```

Or directly:

```
AnimatedSprite_play((AnimatedSprite)this->sprite, this-
>characterDefinition->animationDescription, animationName);
```