

Vircon32: How to make games

Document date 2023.12.15

Written by Carra

What is this?

This document is a quick guide to start making games for the Vircon32 virtual console. The goal is to start from scratch and take a series of steps to know how to create very basic games.

Summary

This guide is organized in small sections, that first teach us the process at an overall level and later focus on each of the individual aspects that a game needs: image, sound, player control, etc.

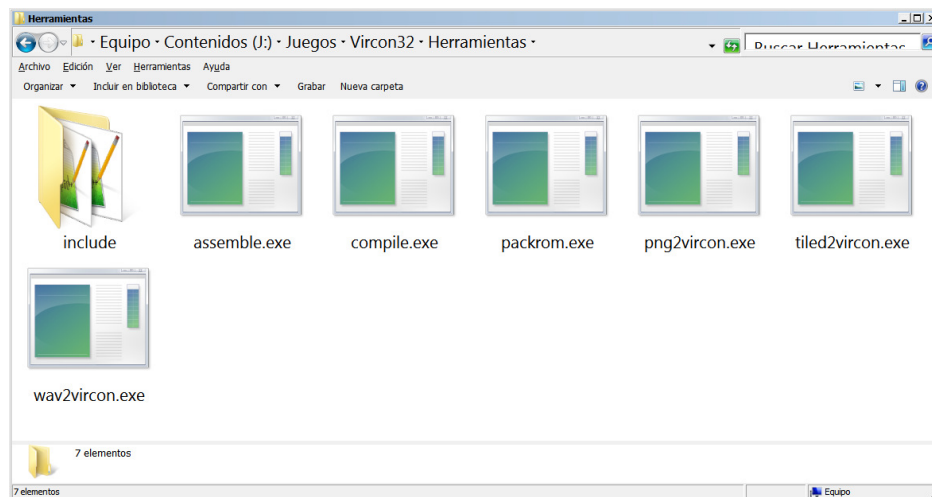
Introduction.....	2
Development tools	2
Packing a ROM	4
Automating the process	5
Typical game structure.....	6
Interpreting compiler errors.....	8
Reading the state of gamepads.....	9
Drawing characters and objects.....	9
Scrolling backgrounds	11
Sound effects	11
Background music.....	12
Handling game objects.....	12
Where to keep learning	13

Introduction

This guide focuses mainly on the actual process of writing a C program and using the development tools to run it on the console. We will also show typical examples of how to use some console functions, but in this guide we will not go into full detail about everything that can be done in Vircon32. To better understand the console systems it is recommended to read the guide on how it works.

Development tools

When we download the development tools for Vircon32, we will find a folder containing the following elements:



All these executables are command line programs that we can automate either by writing scripts or by using these tools from other programs such as a development environment (IDE). To do this we will usually need to add the DevTools folder to the Windows executable paths, using the PATH variable.

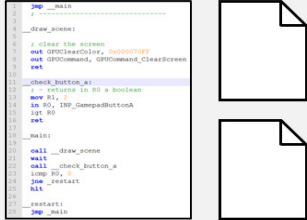
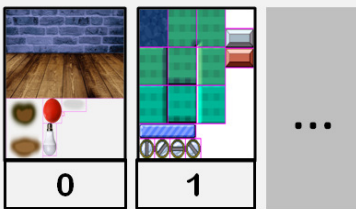
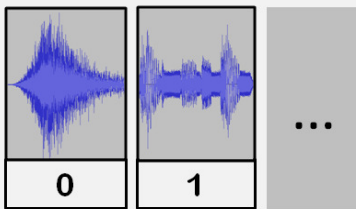
Let's see what each of these elements is:

- **compile.exe** is the C compiler. It translates our C language programs into assembly code.
- **assemble.exe** is the assembler. It converts the assembly code to binary instructions that the Vircon32 CPU can understand.
- **packrom.exe** is the rom packer. It takes the binary program and adds to it the images and sounds it needs to use. The result is a single .v32 file that the Vircon emulator can now load.
- **png2vircon.exe** converts images in PNG format to the internal format that the Vircon32 graphics chip can use.
- **wav2vircon.exe** converts sounds in WAV format to the internal format that can be used by the Vircon32 sound chip.

- **tiled2vircon.exe** is a tool that imports tile maps created in the Tiled editor. It converts each layer into a binary file that we can include in the cartridge to use it as a 2D array.
- **include:** this folder contains the files for the C standard library that the compiler and programs will use.

Step to create a program

The Vircon32 console works with virtual cartridges which, same as in other emulators, are rom files. Each game is a single file in which everything the game needs is packed. This content is divided into 3 separate parts:

Program ROM	Video ROM	Audio ROM
<p>Instructions + Data</p> 	<p>List of images</p> 	<p>List of sounds</p> 

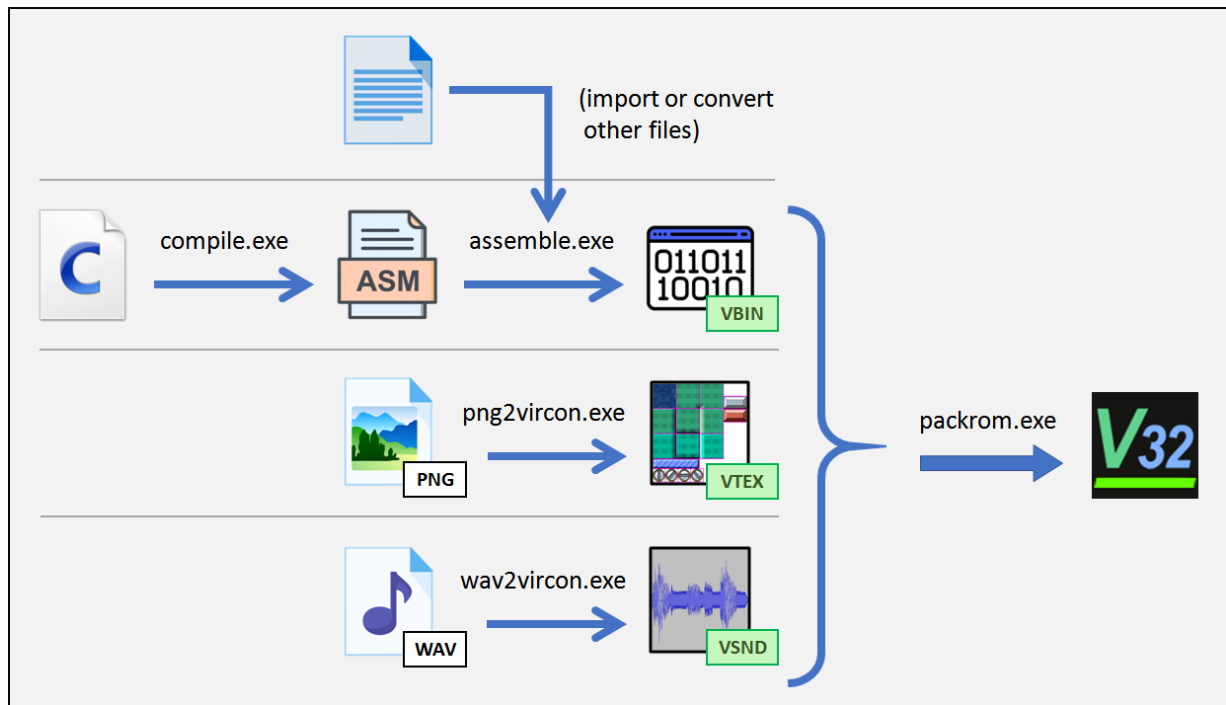
If Vircon32 cartridges only included the program, creating a Vircon32 game from the code would only involve these 2 steps:



However, in Vircon32 with this we would only have created the program rom. We will also need to include in the cartridge a series of images and sounds so that the program can use them. In general the steps to follow are the following:

- 1) Compile and assemble our C program
- 2) Convert all images to the native format of Vircon32
- 3) Convert all sounds to the native format of Vircon32
- 4) Use the packer to join everything into the final rom file
- 5) Use the emulator to test our program

We may also have some additional data files (such as tile maps) that we need to import. In general the process follows this model:



The files marked in green are those that are already in native Vircon32 formats, and therefore can be included in the final packing.

Packing a ROM

The packer may need to include many files in the rom (a Vircon32 game can have up to 256 images and 1024 sounds). So providing it with the file paths via command line would not be very practical.

Instead it uses as input parameter an XML file containing the complete definition of the rom, including also some general details like title and version. In the examples you can see how it is used, but the form of the XML document is as seen here:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<rom-definition version="1.0">
  <rom type="cartridge" title="Game title" version="1.0" />
  <binary path="PathToBinary.vbin" />
  <textures>
    <texture path="PathToTexture1.vtex" />
    <texture path="PathToTexture2.vtex" />
  </textures>
  <sounds>
    <sound path="PathToSound1.vsnd" />
    <sound path="PathToSound2.vsnd" />
  </sounds>
</rom-definition>
```

The order in which the textures and sounds appear is the same order in which they will be packed in the file. That is, our program will be able to access the first texture in the list with ID 0, the second with ID 1, and so on.

A program can have no textures and no sounds: the program binary is the only one of the 3 parts that is mandatory. However the XML must always have the `<textures>` and `<sounds>` elements even if they are empty.

Automating the process

The packer only does the last step to create our game, with the program already compiled and the sounds and images already converted. Creating a game means creating our rom again every time we make changes, so we usually want to use some script or external tool to automate it all with a single click.

The easiest way is to write a small command line script to run the steps. Practical examples can be found in the included program sources, but we will show here this simple example of a Windows BAT command file.

```
@echo off

echo Compile the C code
echo -----
compile Program.c -o obj\Program.asm || goto :failed

echo Assemble the ASM code
echo -----
assemble obj\ Program.asm -o obj\Program.vbin || goto :failed

echo Convert the PNG textures
echo -----
png2vircon Texture.png -o obj\Texture.vtex || goto :failed

echo Convert the WAV sounds
echo -----
wav2vircon Sound.wav -o obj\Sound.vsnd || goto :failed

echo Pack the ROM
echo -----
packrom RomDefinition.xml -o bin\FinalGame.v32 || goto :failed
goto :succeeded

:failed
echo BUILD FAILED
exit /b %errorlevel%

:succeeded
echo BUILD SUCCESSFUL
exit /b

@echo on
```

This script could be simplified, but the important thing is that it calls all the stages and, if an error happens in any of them, it stops the process instead of trying to continue.

Something that we can also see in this script is that it uses the subdirectories `obj` and `bin`. Programming in C (not only in Vircon32) it is usual to use a folder structure like this for every project. This way we can separate the final result (in `"bin"`), the intermediate files that are being generated (in `"obj"`), and the project's source files themselves.

Typical game structure

We already know how to create the game from the C program. Now we'll see how to write the program itself. This is not intended to teach programming to those who have never done it, nor is it a full guide on how to program games. But we will give a few basic notions with examples, and see some usually followed practices when coding games.

In general, a Vircon32 game is usually written following this structure:

- Include libraries
- Give a name to our textures, regions y sounds
- Declare global variables
- Auxiliary functions
- **Main function**
 - Configure textures and sounds
 - Initialize the game
- **Main loop**
 - Read the gamepads and apply player's actions
 - Simulate the mechanics of our game
 - Draw on screen the scene and the objects in it
 - Wait for the next frame to control game speed

The most important element in this structure is the main loop. Normally games are always repeated in an infinite loop, and the actions in this loop are performed once per frame. In Vircon32 this means that we repeat it 60 times per second.

On the next page we can see a small complete example that follows this program structure. This program is very simple (it only draws the character on a fixed background and allows us to move it), but it allows us to see on a single page the structure in practice. Additionally, the different parts are marked with comments.

```

// include Vircon libraries
#include "video.h"
#include "input.h"
#include "time.h"

// -----
// DEFINITIONS

// give names to our texture regions
#define RegionBackground 0
#define RegionRobot      1

// -----
// MAIN FUNCTION

void main( void )
{
    // -----
    // PART 1: CONFIGURE OUR TEXTURES

    select_texture( 0 );

    // define our texture regions
    select_region( RegionBackground );
    define_region_topleft( 0,0, 639,359 );
    select_region( RegionRobot );
    define_region( 1,361, 66,441, 33,417 );

    // -----
    // PART 2: INITIALIZATIONS

    // our robot starts at the screen center
    int RobotX = screen_width / 2;
    int RobotY = screen_height / 2;

    // -----
    // PART 3: MAIN LOOP

    // keep repeating our game logic for every frame (60 fps)
    while( true )
    {
        // move robot in the direction we press
        int DirectionX, DirectionY;
        gamepad_direction( &DirectionX, &DirectionY );
        RobotX += 2 * DirectionX;
        RobotY += 2 * DirectionY;

        // draw our background to fill the screen
        select_region( RegionBackground );
        draw_region_at( 0, 0 );

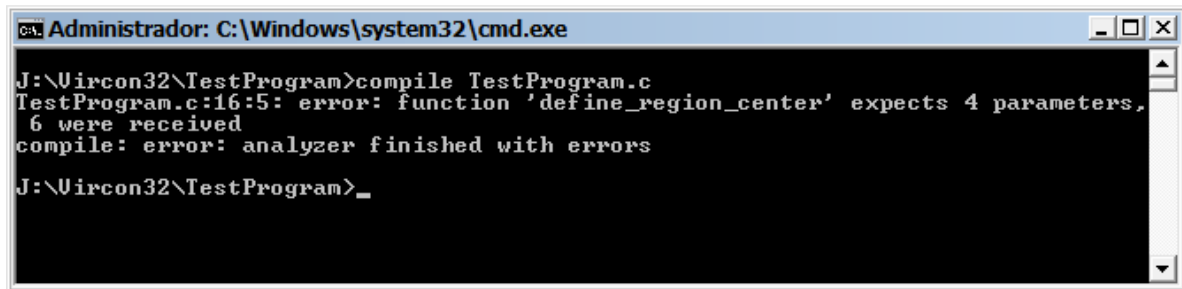
        // now draw robot in its current position
        select_region( RegionRobot );
        draw_region_at( RobotX, RobotY );

        // wait for next frame to control game speed
        end_frame();
    }
}

```

Interpreting compiler errors

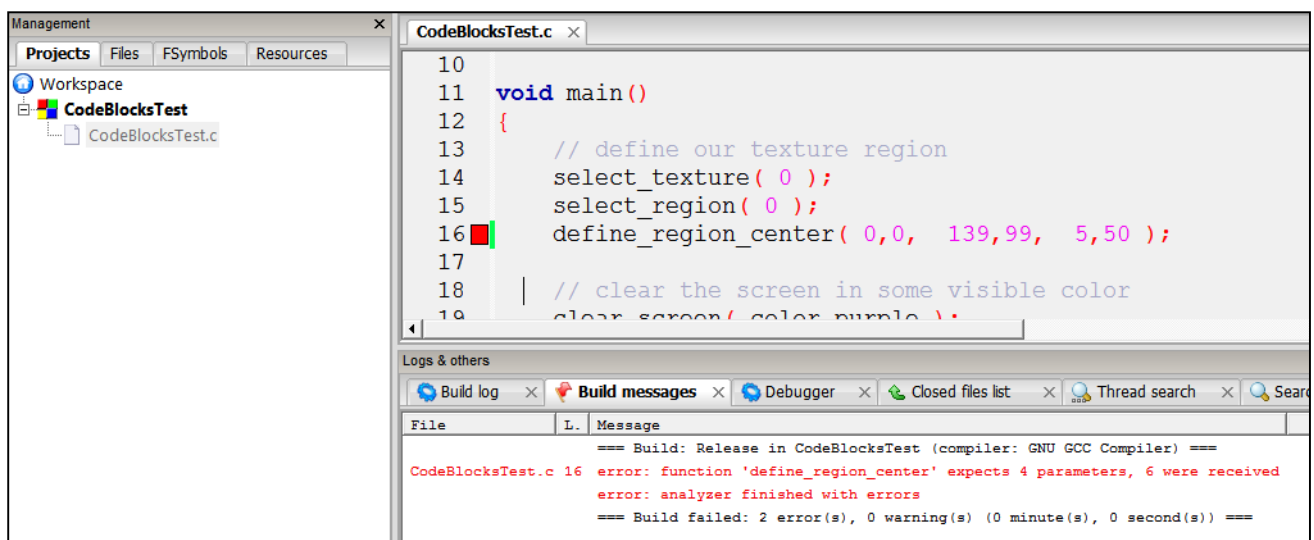
It is normal for us to make mistakes in our programs, so we will be interested in knowing how to find an error when the compiler warns us. The Vircon32 C compiler outputs errors in the same format as the gcc compiler, that is: file name, line and column. You can see how it looks in this example:



```
Administrator: C:\Windows\system32\cmd.exe
J:\Vircon32\TestProgram>compile TestProgram.c
TestProgram.c:16:5: error: function 'define_region_center' expects 4 parameters,
6 were received
compile: error: analyzer finished with errors
J:\Vircon32\TestProgram>_
```

If we program without an IDE we can look for these errors manually with a text editor that includes line numbers, such as for instance Notepad++. However, since the compiler is designed to behave like gcc (to some extent), it is possible to work with an IDE by configuring it correctly.

Among the example sources that you can download, there is one called "CodeBlocksTest", which is precisely a project already created for Code::Blocks. With it we can not only compile our program but also use the Run button to test the rom in the emulator. In this image you can see how Code::Blocks will show us the same error.



Although this method has not been tested in other IDEs, it is quite possible that it can work with them as well. What has been done in this example project is to configure our C files so that Code::Blocks does **NOT** compile them. Instead, we tell the IDE to run our BAT file (like the one shown previously) as a post-compilation event.

And with this we already have an idea of how to work with a program at a global level. What we will do in the following sections is to talk about some actions that are normally needed in a game, and show ways in which we can accomplish them.

Reading the state of gamepads

The first thing we usually need to do in the main loop is to know what the player is pressing on the gamepad to update the game. Here we show 2 alternatives: check individual directions or check them globally.

```
// here we move only in X, ignoring the vertical directions
if( gamepad_left() > 0 ) Position.x -= Speed;
if( gamepad_right() > 0 ) Position.x += Speed;

// here we move in any direction automatically
gamepad_direction( &DirectionX, &DirectionY );
Position.x += Speed * DirectionX;
Position.y += Speed * DirectionY;
```

What happens with the buttons? In Vircon32 there are no events. To know whether a button was already pressed or has just been pressed right now, we would normally need to save the state of the previous frame and see if it changed. However this is not necessary: the console already gives us this information. We can detect it like this:

```
// read the state of button A
int ButtonA = gamepad_button_a();

// when we press A we begin charging energy
if( ButtonA == 1 )
    StoredEnergy = 1;

// for each frame it remains pressed, energy increases
else if( ButtonA > 0 )
    StoredEnergy++;

// when button is released, we shoot
if( ButtonA == -1 )
    Shoot();
```

Drawing characters and objects

It is common in a game that characters can have animations. A simple method for that is to define several consecutive regions and cycle through all of them according to the elapsed frames.

Let's suppose for example that we have our character drawn like this:



The C libraries allow us to define these consecutive regions all at once. In this example we can see how this is done, and how we would animate it when drawing:

```
// define the regions for our animation
define_region_matrix
(
    100,    // ID of the first region to create
    1,1,    // top-left corner for the first region
    50,50,   // bottom-right corner for the first region
    25,47,   // reference point for the first region
    7,1,     // the matrix has 7 regions in X and 1 in Y
    1       // regions have a separating border of 1 pixel
);

// change animation image every 5 frames
AnimationTime++;

if( AnimationTime % 5 == 0 )
    AnimationImage++;

// when animation ends, it will repeat from the beginning
if( AnimationImage >= 7 )
    AnimationImage = 0;

// each frame we draw our animation
select_region( 100 + AnimationImage );
draw_region_at( PlayerX, PlayerY );
```

In many games characters are not drawn facing both sides, but instead are always drawn facing right. Then, to make them face left the image is flipped. We can achieve this with a scaling in X equal to -1.

```
// draw character facing right
if( PlayerSpeedX >= 0 )
    draw_region_at( PlayerX, PlayerY );

// draw character facing left
else
{
    set_drawing_scale( -1, 1 );
    draw_region_zoomed_at( PlayerX, PlayerY );
}
```

Scrolling backgrounds

The textures used by Vircon32 (1024x1024 pixels) are larger than the screen (640x360), but even then they will be too small if we want to make large scenarios. The console allows us to use several textures and connect them, but the most used method in many games is to create tile maps.

Among the programs included in DevTools is `tilted2vircon.exe`. This program allows us to import tile maps made with the editor Tiled. The way to do this is to call our importer tool before compiling the C program, so that at compilation the imported file is already created. Let's say that using the import tool we created a file called "Level1.map" in the obj folder of the project. Then from C we can import this into our program as if it were an array like this:

```
// the contents of the file will be saved as an array in the cartridge  
embedded int[ TilesInY ][ TilesInX ] MapBricks = "obj\\Level1.map";
```

Then, in our textures we will have the images of each tile with which to draw the map. To define the set of tiles in the program we would use function `define_region_matrix()` just as before.

After that, to draw the map on the screen, we can use a loop that walks through the map in X and Y. But be careful! A map could be very large, and trying to draw all of it every frame can demand too much from the console. In that case we should check what range of tiles are visible in the screen and limit the loop to that range in X and Y.

Sound effects

In Vircon32 there exist 16 channels to play sounds. We have the option to play automatically (the function will search for a free channel).

```
// we play a sound in any free channel  
play_sound( SoundExplosion );
```

Another option is to manually reserve some sound channels. For example, if we have only one channel for our character to speak, we make sure that 2 phrases can never be playing at the same time (which would sound weird).

```
// stop any previous character sound  
if( get_channel_state( ChannelPlayer ) != channel_stopped )  
    stop_channel( ChannelPlayer );  
  
// now we can already play a new sound  
play_sound_in_channel( SoundHello, ChannelPlayer );
```

Background music

Normally we will be interested in reserving a certain known channel for the music. This way we can pause or resume the music when needed. It is also common for the music to play continuously and repeat in a loop. Here we show both things:

```
// configure this music to be played in a loop
select_sound( MusicLevel1 );
set_sound_loop( true );

// now our music will play with loop enabled
play_sound_in_channel( MusicLevel1, ChannelMusic );

// stop the music channel
stop_channel( ChannelMusic );
```

Handling game objects

When you have a more complex game, there can be a large variety of objects that need to be updated and drawn every frame: player, shots, various types of enemies, etc. In C++ it is usual to use class inheritance to handle these game entities in a more generic way. In C we can use unions to achieve that. In this small example we see how we can define objects with variable data:

```
// define structures with different data
// depending of what each object needs
struct EnemyState{ ... };
struct BulletState{ ... };

// this union can store data for any of our objects
union ObjectState
{
    EnemyState AsEnemy;
    BulletState AsBullet;
};

// we add information to the union to know what it actually contains
struct GameObject
{
    bool Active;
    int ObjectType;
    ObjectData State;
}
```

Then we must see how to handle that data. Currently the Vircon32 compiler is still limited and there is still no support for dynamic memory (that means we cannot use lists), nor pointers to functions. However we can handle our objects at a more generic level using an array as we show here:

```

// we use an array with the maxium amount of object that can exist
GameObject[ 50 ] AllObjects;

// we would process the objects like this
for( int i = 0; i < 50; i++ )
{
    GameObject* ThisObject = GameObject[ i ];

    if( !GameObject[ i ]->Active )
        continue;

    // choose the type of object
    if( ObjectType == TypeEnemy )
        GameObject[ i ]->Active = ProcessEnemy( GameObject[ i ]->AsEnemy );

    else if( ObjectType == TypeBullet )
        GameObject[ i ]->Active = ProcessBullet( GameObject[ i ]->AsBullet );
}

```

Where to keep learning

With this we have seen the basics of what is needed to create a game, but there is still a much more to learn. To practice what we have seen here, you can start by compiling the programs from the tutorials and experimenting with them.

Later on, a good way to advance and think of new ideas is to see how other games and programs for Vircon32 have been made. The source code of the test programs that are available can serve as more advanced examples, as they also have quite a lot of comments and usually follow the same structure shown here.

It is also recommended for the future to know in detail how the different systems of the console work. This is explained in the corresponding guide of this same document set.