Virtual Game Console

Mikael Thieme
mikael.thieme@his.se

School of Informatics
University of Skövde, Sweden

1 Introduction

Virtual Game Console (VGC) is a C++ library that aims to provide a simple game programming environment for modern operating systems. In essence, the library achieves this by insulating the programmer from the details surrounding window management and event-based programming in these systems. As suggested by the name, the resulting environment resembles older game consoles in that it involves a fixed sized display and allows for poll-based (as opposed to event-based) access to buttons etc.

VGC uses only the most basic features of C++ which means that a novice programmer should be able to understand this document and begin using the library after only a few introductory C++ lectures or after reading the first few chapters in a C++ book.

2 Tutorial

In this chapter you will learn how to write a basic “hello world” program in VGC. To begin with you need to download and unpack the library zip-file which is available from the course page in SCIO.

- Step 1. Open the visual studio solution for VGC by double clicking on the solution (.sln) file in windows explorer:

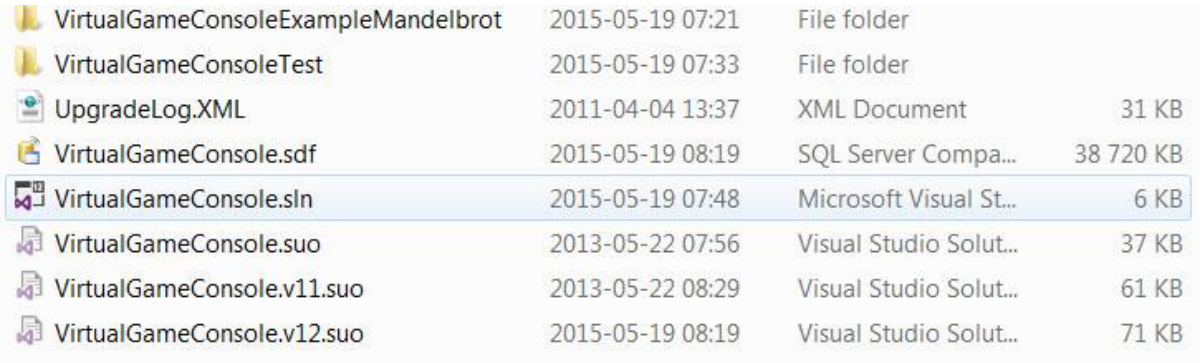


Figure 1. The VGC solution file in windows explorer.

The solution for VGC consists of several projects shown in the solution explorer in visual studio:

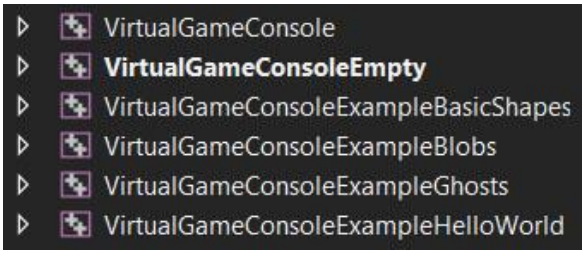


Figure 2. Some of the projects in the VGC solution.

There is one project for the library itself as well as several example projects that use the library project. There is also an empty project which is meant to be used for your own projects and assignments. The empty project by default selected as the startup project. This means that this project will compile and run when you press the play button in visual studio:

Figure 3. Pressing the play button in visual studio compiles and runs the current startup project.

- Step 2. Press play to compile and run the empty project. If everything is working correctly you should be able to follow the compilation process in the output window:

```
1>------ Build started: Project: VirtualGameConsole, Configuration: Debug Win32 ------
1>  VGCWindow.cpp
1>  VGCVirtualGameConsole.cpp
1>  VGCVector.cpp
1>  VGCTimerImp.cpp
1>  VGCReferenced.cpp
1>  VGCRectangle.cpp
1>  VGCRandomizer.cpp
1>  VGCMessage.cpp
1>  VGCKeyboard.cpp
1>  VGCKey.cpp
1>  VGCImageImp.cpp
1>  VGCHandler.cpp
1>  VGCGDIPlus.cpp
1>  VGCFontImp.cpp
1>  VGCError.cpp
1>  VGCDisplay.cpp
1>  VGCColor.cpp
1>  VGCClock.cpp
1>  VGCBus.cpp
1>  VGCBootstrap.cpp
1>  Generating Code...
1>  Compiling...
1>  VGCAssert.cpp
1>  VGCAdjustment.cpp
1>  Generating Code...
1>  VirtualGameConsole.vcxproj -> C:\Work\VirtualGameConsole\VirtualGameConsole\Debug\VirtualGameConsole.lib
2>------ Build started: Project: VirtualGameConsoleEmpty, Configuration: Debug Win32 ------
2>  Main.cpp
2>  VirtualGameConsoleEmpty.vcxproj -> C:\Work\VirtualGameConsole\Debug\VirtualGameConsoleEmpty.exe
========== Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Figure 4. The compilation process shown in the output window. In this case the basic library and the empty project is compiled in order (since the latter depends on the former).

Since the empty project is meant for your own projects it does not do anything interesting by default. The project essentially consists of an empty main function which hence immediately terminates the program:

```
'VirtualGameConsoleEmpty.exe' (Win32): Loaded 'C:\Windows\SysWOW64\cryptbase.dll'. Cannot find or open the PDB file.
'VirtualGameConsoleEmpty.exe' (Win32): Loaded 'C:\Windows\SysWOW64\imm32.dll'. Cannot find or open the PDB file.
'VirtualGameConsoleEmpty.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msctf.dll'. Cannot find or open the PDB file.
'VirtualGameConsoleEmpty.exe' (Win32): Loaded 'C:\PROGRA~2\Sophos\SOPHOS~1\SOPHOS~1.DLL'. Cannot find or open the PDB file.
'VirtualGameConsoleEmpty.exe' (Win32): Loaded 'C:\Windows\SysWOW64\psapi.dll'. Cannot find or open the PDB file.
The program '[296] VirtualGameConsoleEmpty.exe' has exited with code 0 (0x0).
```

Figure 5. The startup of the empty project shown in the output window. The executable (.exe) file loads several windows library (.dll) files then terminates.

- Step 3. Change the current project to the ghost example by right clicking on the project and choose “Set as StartUp Project”:

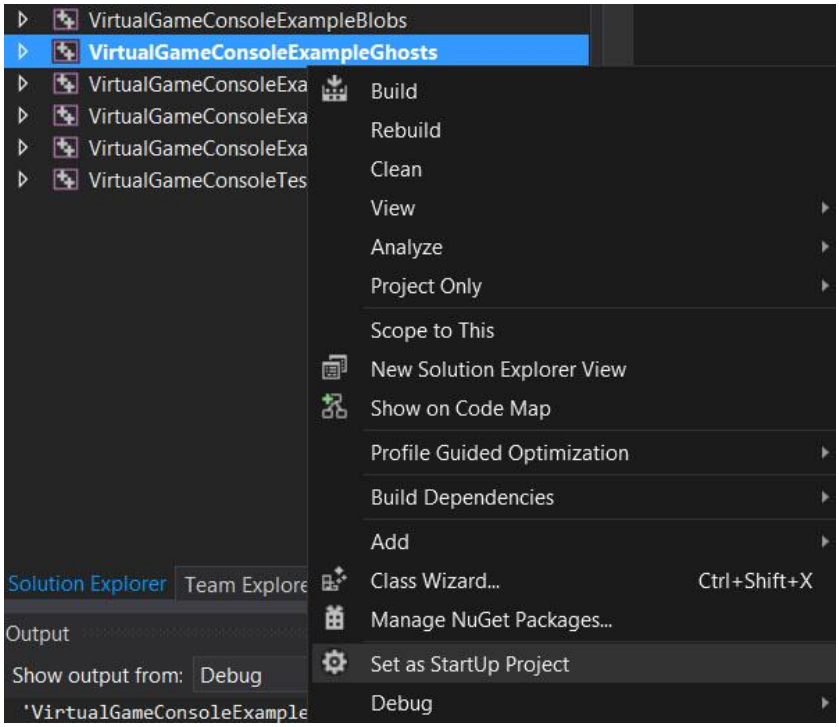


Figure 6. Changing the current project to the ghost example.

- Step 4. Press play again to run the ghost example. If everything works correctly you should now see a few animated ghosts bouncing around in the application window:

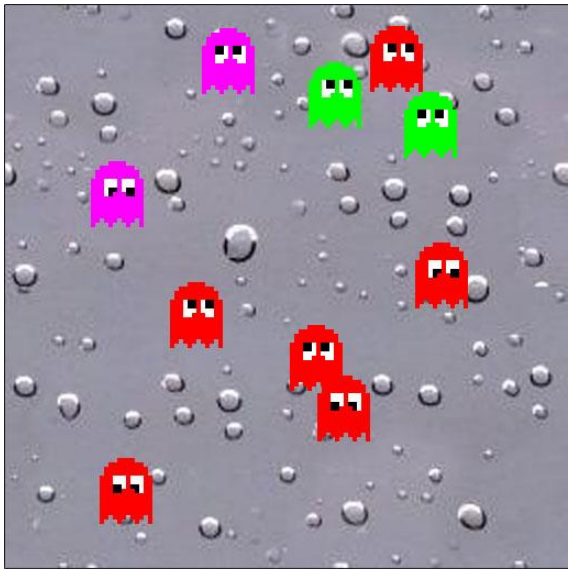
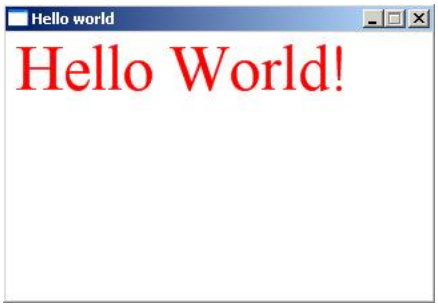


Figure 7. The ghost example project.

- Step 5. Terminate the example program by pressing the close ("X") button:

- Step 6. Try out some of the other example projects in a similar manner. Especially you should try the hello world example which is a finished version of the hello world program that you will create below. Running the hello world example gives the following output in the application window:



- Figure 8. Hello world!

In the following steps you will learn how to write this program in VGC. Most of these steps are used in all VGC projects so much of what you learn here will also be useful later on in the course.

- Step 7. Reset the current startup project to be the empty project.
- Step 8. Open the source code file (Main.cpp) double clicking on it in the solution explorer:

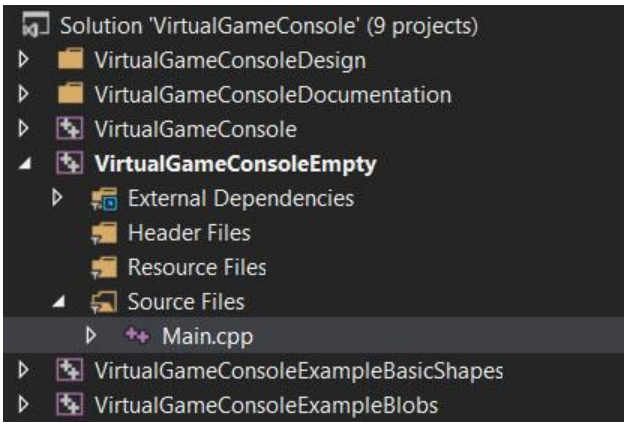


Figure 9. Open the source code file by double clicking on it in the solution explorer.

You should now be able to see and edit the program code for the empty project. Initially the program consists of the following lines:

```
#include "VGCVirtualGameConsole.h"

int VGCMain(const VGCStringVector &arguments){
     return 0;
}
```

The first line informs the compiler that we want to use the VGC library. By including this file you ensure that all types and functions in VGC are declared and hence that they can be used in your program.

The remaining lines defines the VGCMain() function which constitutes starting point in all VGC programs. The return value from this function indicates whether or not the program has completed successfully (0 indicates success, all else indicate failure).

VGCMain() takes a single parameter that stores the command line arguments in the form of a vector of strings. Lets say, for example, that the program is started from a command prompt using the following line.

```
helloworld.exe argument1 argument2
```

Then the following conditions are true.

```
arguments.size() == 2
arguments[0] == ”argument1”
arguments[1] == ”argument2”
```

If no command line arguments are used, as is the case below, this parameter can simply be ignored.

Before we can use VGC fully the library must be initialized by calling the VGCVirtualGameConsole::initialize() function. Similarly, before exit, the program should finalize VGC by calling VGCVirtualGameConsole::finalize(). The rest of the program will later be inserted between these two calls. Extend the program as follows. New lines are marked in bold face.

```
#include "VGCVirtualGameConsole.h"
#include <string>

using namespace std;

int VGCMain(const VGCStringVector &arguments){

     const string applicationName = "Hello world";
     const int    DISPLAY_WIDTH   = 320;
     const int    DISPLAY_HEIGHT  = 200;
     VGCVirtualGameConsole::initialize(applicationName, DISPLAY_WIDTH,
DISPLAY_HEIGHT);


     VGCVirtualGameConsole::finalize();

     return 0;
}
```

The first couple of new lines inform the compiler that we want to use the C++ standard type for text strings in our program. (The details of this will not be discussed here, but the reader is encouraged to look it up elsewhere, e.g. in the course literature.)

The next group of statements initialize VGC by specifying the application name, display width, and display height. The application name will appear on the title bar of the resulting window which has a client area of the specified width and height (320*200 pixels in this case).

Finally, VGCVirtualGameConsole::finalize() is called to finalize VGC.

Compile and run the program to ensure that things are working fine so far. You should be able to see the window appear for a fraction of a second when running the program.

Next we add some code to open a font that we will later use to render the text string. Also, at the same time add a line of code to close the font before finalizing VGC.

```
#include "VGCVirtualGameConsole.h"
#include <string>

using namespace std;

int VGCMain(const VGCStringVector &arguments){

     const string applicationName = "Hello world";
     const int    DISPLAY_WIDTH   = 320;
     const int    DISPLAY_HEIGHT  = 200;
     VGCVirtualGameConsole::initialize(applicationName, DISPLAY_WIDTH,
DISPLAY_HEIGHT);

     const int FONT_SIZE = 48;
     VGCFont   font      = VGCDisplay::openFont("Times New Roman", FONT_SIZE);


     VGCDisplay::closeFont(font);

     VGCVirtualGameConsole::finalize();

     return 0;
}
```

The example uses the Times New Roman font of size 48 which roughly corresponds to the height of the characters in pixels.

At this point we are ready to add the main loop to the program. In a computer game this loop is typically called the game loop and it consists of three major stages.

- Step 1. Receive input from the user
- Step 2. Update game objects
- Step 3. Render game objects

Our simple program will only need the third step since the text string is static and does not respond in any way to input.

Anyhow, you need to signal the beginning and end of the game loop by calling VGCVirtualGameConsole::beginLoop() and VGCVirtualGameConsole::endLoop() respectively. If VGCVirtualGameConsole::beginLoop() returns true the game loop may execute, otherwise the application should exit. Primarily the latter situation arises when the user closes the application window. In the light of this, add code as follows.

```
#include "VGCVirtualGameConsole.h"
#include <string>

using namespace std;

int VGCMain(const VGCStringVector &arguments){

     const string applicationName = "Hello world";
     const int    DISPLAY_WIDTH   = 320;
     const int    DISPLAY_HEIGHT  = 200;
     VGCVirtualGameConsole::initialize(applicationName, DISPLAY_WIDTH,
DISPLAY_HEIGHT);

     const int FONT_SIZE = 48;
     VGCFont   font      = VGCDisplay::openFont("Times New Roman", FONT_SIZE);

     while(VGCVirtualGameConsole::beginLoop()){
          VGCVirtualGameConsole::endLoop();
     }

     VGCDisplay::closeFont(font);

     VGCVirtualGameConsole::finalize();

     return 0;
}
```

In a similar way, you need to inform VGC about the beginning and end of the rendering step of the game loop. This is done by calling VGCDisplay::beginFrame() and VGCDisplay::endFrame respectively:

```
#include "VGCVirtualGameConsole.h"
#include <string>

using namespace std;

int VGCMain(const VGCStringVector &arguments){

     const string applicationName = "Hello world";
     const int    DISPLAY_WIDTH   = 320;
     const int    DISPLAY_HEIGHT  = 200;
     VGCVirtualGameConsole::initialize(applicationName, DISPLAY_WIDTH,
DISPLAY_HEIGHT);
```

```
	const int FONT_SIZE = 48;
	VGCFont   font      = VGCDisplay::openFont("Times New Roman", FONT_SIZE);

	while(VGCVirtualGameConsole::beginLoop()){

		if(VGCDisplay::beginFrame()){
			VGCDisplay::endFrame();
		}

		VGCVirtualGameConsole::endLoop();
	}

	VGCDisplay::closeFont(font);

	VGCVirtualGameConsole::finalize();

	return 0;
}
```

Next we can add the final lines of code in order to fill the window with a white background color and render the text string:

```
#include "VGCVirtualGameConsole.h"
#include <string>

using namespace std;

int VGCMain(const VGCStringVector &arguments){

	const string applicationName = "Hello world";
	const int    DISPLAY_WIDTH   = 320;
	const int    DISPLAY_HEIGHT  = 200;
	VGCVirtualGameConsole::initialize(applicationName, DISPLAY_WIDTH,
DISPLAY_HEIGHT);

	const int FONT_SIZE = 48;
	VGCFont   font      = VGCDisplay::openFont("Times New Roman", FONT_SIZE);

	while(VGCVirtualGameConsole::beginLoop()){

		if(VGCDisplay::beginFrame()){

			const VGCColor backgroundColor = VGCColor(255, 255, 255, 255);
			VGCDisplay::clear(backgroundColor);

			const string        text           = "Hello World!";
			const VGCColor      textColor      = VGCColor(255, 255, 0, 0);
			const VGCVector     textPosition   = VGCVector(0, 0);
			const VGCAdjustment textAdjustment = VGCAdjustment(0.0, 0.0);
			VGCDisplay::renderString(
				font,
				text,
				textColor,
				textPosition,
				textAdjustment);

			VGCDisplay::endFrame();
		}

		VGCVirtualGameConsole::endLoop();
	}

	VGCDisplay::closeFont(font);
```

```
        VGCVirtualGameConsole::finalize();

        return 0;
}
```

We will not dive into the details of these lines of code here but the reader is encouraged to experiment with the program to determine the meaning of text adjustment, how colors are represented etc. It is not important, at this stage, to understand all details of this program, as this will be described in upcoming chapters. Make sure, however, before continuing, that you understand the basic structure of the program, i.e. initialization, finalization, and the structure of the game loop.

3 Components

VGC consists of several components organized as illustrated in Figure 10. Each component is implemented in separate header (.h) and implementation (.cpp) files. The curious reader is encouraged to browse through these files to get a feel for their content. (To be perfectly honest VGC consists of several more components but these are only used internally by VGC.). Each arrow in the figure indicates a "depends on" relation. This implies, among other things, that in order to understand a higher level component fully, one must first understand the corresponding lower level components.

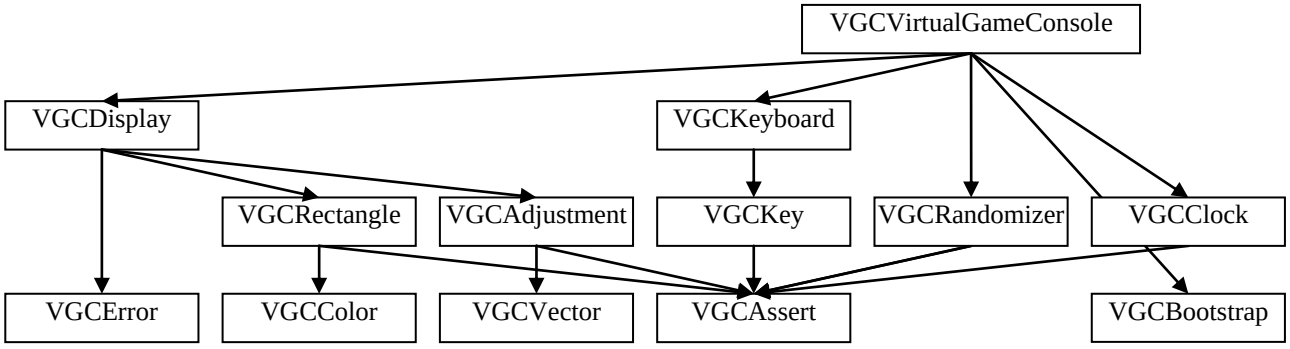


Figure 10. Components in VGC.

The following subsections describe each of these components in detail and in bottom up order.

3.1 VGCError

VGCError defines a macro for displaying error messages on the screen.

```
VGCError(<description>)
```

The macro is used internally by VGC to report misuse or other errors, but may also be used by clients of VGC to report errors at the application level. Along with the description, this macro also outputs the source file name and row number that contains the call.

Lets say, for example, that the following line of code is executed at line 4 of a source file name helloworld.cpp.

```
VGCError("This is an error description.");
```

The output may then look something like in Figure 11.

Figure 11. Error description.

It is also possible to compose more complex error messages by using standard string streams, e.g. to include the value of some integer variable v into the message:

```
#include <sstream>
using namespace std;
.
.
```

```
int v = 1002;
ostringstream description;
description << "The value of v (" << v << ") is no good.";
VGCError(description.str());
.
.
```

The output from this program is shown in Figure 12.

Figure 12. Fancy error description.

3.2 VGCColor

Each color in VGC consists of four components: alpha, red, green, and blue. Each component, in turn, has value ranging from 0 (no intensity) to 255 (full intensity). The red, green, and blue components define the color itself whereas the alpha component defines opacity/transparency. When a non opaque (alpha < 255) color is used to draw a graphical object (e.g. a line) the background is partially visible through the object.

In VGC, variables representing colors are defined using the VGCColor type. For example:

```
VGCColor default;                          //Default color; transparent black
VGCColor black(255, 0, 0, 0);              //opaque black
VGCColor white(255, 255, 255, 255);        //opaque white
VGCColor gray(255, 128, 128, 128);               //opaque gray
VGCColor red(255, 255, 0, 0);              //opaque red
VGCColor green(255, 0, 255, 0);            //opaque green
VGCColor blue(255, 0, 0, 255);             //opaque blue
VGCColor yellow(255, 255, 255, 0);               //opaque yellow
VGCColor turquoise(255, 0, 255, 255);      //opaque turquoise
VGCColor transparentRed(128, 255, 0, 0);         //partially transparent red
.
.
```

Furthermore, it is possible, in VGC, to initialize a color using another color, assign one color to another, and read/set individual color components. For example:

```
/* Initializing one color using another */
VGCColor color1(255, 0, 255, 255);
VGCColor color2(color1);

/* Assigning one color to another */
VGCColor color1(255, 0, 255, 255);
VGCColor color2;
color2 = color1;

/* Setting/reading color components */
VGCColor color;
color.setAlpha(255);
color.setRed(255);
color.setGreen(0);
color.setBlue(0);
bool colorIsOpaque = (255 == color.getAlpha());
```

3.3 VGCVector

The VGCVector type is used in VGC to represent two dimensional vectors each having an x and y component. Typically, a vector refers to a particular coordinate on the display as illustrated in Figure 13.

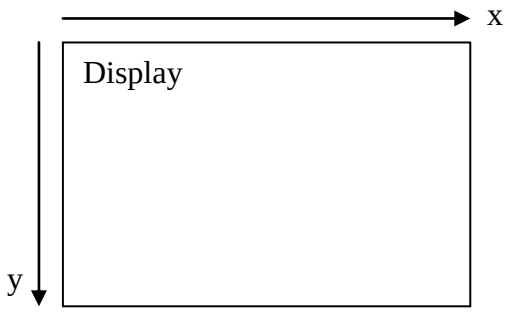


Figure 13. The display area.

Note especially, in this case, that origo is located at the upper left corner of the display and that the positive y-axis is directed downwards on the screen.

Assume, for example, that the display is 320 pixels wide and 200 pixels high. Then the corner pixels of the display are represented like this:

```
VGCVector upperLeft;               //Upper left pixel (default origo)
VGCVector lowerLeft(0, 199);       //Lower left pixel
VGCVector upperRight(319, 0);            //Upper right pixel
VGCVector lowerRight(319, 199);    //Lower right pixel
```

It is of course also possible to represent coordinates outside of the display, for example to represent the position of an alien spacecraft that is not yet visible on the display.

```
VGCVector spacecraftPosition(-2000, 1000);
```

It is possible to initialize a vector using another vector, assign one vector to another, and to read/set each component individually:

```
/* Initialization */
VGCVector vector0(13, 13);
VGCVector vector1(vector0);

/* Assignment */
VGCVector vector0(12, 14);
VGCVector vector1;
vector1 = vector0;

/* reading/setting components */
VGCVector vector;
vector.setX(12);
vector.setY(14);
if((12 == vector.getX()) && (14 == vector.getY())){
     ...
}
```

Furthermore, the operators +, -, *, ==, and != are defined for this type. For example:

```
VGCVector v0(1, 2);
VGCVector v1(3, 4);
VGCVector v2 = v0 + v1 //v2 = (4, 6)
VGCVector v3 = v0 - v1 //v3 = (-2, -2)
VGCVector v4 = 2 * v0; //v4 = (2, 4)
int product = v0 * v1; //product = 1*3 + 2*4 = 11
bool b0 = (v0 == v1);  //b0 = false
bool b1 = (v0 != v1);  //b1 = true
```

Vector addition can, among other things, be used to implement local coordinate systems. Say, for example, that we want a local coordinate system with an origo placed roughly at the center of the display. This can be achieved by adding the position of the origo (160, 100) to each coordinate in the local system.

3.4 VGCAssert

The VGCAssert component defines three macros that can be used to ensure/assert that a condition is true:

```
VGCAssert(<condition>);
VGCReleaseAssert(<condition>);
VGCDebugAssert(<condition>);
```

If the condition is true program execution will continue normally, but if the condition is false, the program will output an error message displaying the condition together with the filename and line number of the corresponding assertion.

The debug and release versions are only active in respectively debug (preprocessors symbol NDEBUG undefined) and release (preprocessors symbol NDEBUG defined) configurations. The basic VGCAssert() version is active in all configurations. Out of these, prefer the latter version as long as other issues, such as code size and speed, are not critical.

Assertions are used a lot internally in VGC in order to detect misuse. For example, if a program attempts to initialize VGC using a negative display width this will cause an internal assertion to fire as illustrated in Figure 14.

```
const string applicationName = "Hello world";
const int    DISPLAY_WIDTH   = -320;
const int    DISPLAY_HEIGHT  = 200;
VGCVirtualGameConsole::initialize(applicationName, DISPLAY_WIDTH, DISPLAY_HEIGHT);
```

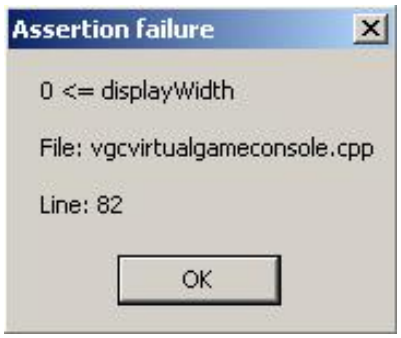


Figure 14. Error description.

Similarly, client code can contain assertions to ensure that otherwise implicit assumptions actually hold during execution. For example, if damage is represented in a game using non negative integers, prefer:

```
void playerWasHit(int damage){
     VGCReleaseAssert(0 <= damage);
     ...
}
```

Note also how the assertion acts like an active comment in the code.

Finally, and most importantly, program correctness should never depend on code within an assert statement. In essence, this means that statements that have side effects should be avoided in assert statements. For example, avoid assert statements like:

```
VGCDebugAssert(0 < i++);
VGCDebugAssert(deleteSomeFile());
```

Note that this recommendation applies to the VGCAssert() version of the macro as well, since it is not uncommon, during optimization, to change from this version to the debug version.

3.5 VGCBootstrap

The VGCBootstrap component defines the main() function and hence constitutes the starting point for the execution of a VGC program. (To be perfectly honest it actually defines the WinMain function that constitutes entry point for Win32 Windows applications.) The main purpose of this component is to extract any command line arguments and in turn call the VGCMain() function. Also, this function will catch any exceptions and output their descriptions on the screen.

The header file for VGCBootstrap further defines the VGCStringVector type used to store any command line arguments.

3.6 VGCRectangle

The VGCRectangle type defined in the component having the same name is used to represent rectangles. A rectangle is defined by integers representing position, width, and height respectively. Typically, but not necessarily, as we shall see below, the position corresponds to the upper left corner of the rectangle, as illustrated in Figure 15.

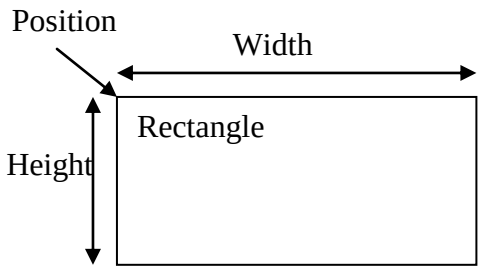


Figure 15. Rectangle properties.

Commonly, rectangles are used to refer to a rectangular area of pixels on the display. Note that such an area may contain no pixels if either the width or height is set to zero.

Suppose, for example, a display of 320*200 pixels, then we can refer to different rectangular portions thereof:

```
/* Default, empty rectangle at position (0, 0)*/
VGCRectangle defaultRectangle;

/* Upper half portion of the display*/
VGCRectangle upperHalf(VGCVector(0, 0), 320, 100);

/* Lower half portion of the display*/
VGCRectangle upperHalf(VGCVector(0, 100), 320, 100);
```

It is possible to (i) initialize one rectangle with another, (ii) assign one rectangle to another, and (iii) set/read individual rectangle properties:

```
/* Initialization */
VGCRectangle r0(VGCVector(1,1), 2, 2);
VGCRectangle r1(r0);

/* Assignment */
VGCRectangle r0(VGCVector(1,1), 2, 2);
VGCRectangle r1;
VGCRectangle r1 = r0;

/* Properties */
VGCRectangle r;
r.setPosition(VGCVector(10, 10));
r.setWidth(2);
r.setHeight(2);
int area = r.getWidth() * r.getHeight();
```

Furthermore, equality (==) and inequality (!=) are defined for rectangles. In VGC, two rectangles are defined to be equal if they have the same porition, width, and height. For example:

```
VGCRectangle r0(VGCVector(1,2), 2, 3);
VGCRectangle r1(VGCVector(2,2), 2, 3);
bool b0 = (r0 == r1);                    //b0 = false
bool b0 = (r0 != r1);                    //b0 = true
```

3.7 VGCAdjustment

The VGCAdjustment type defined by the corresponding component is used to adjust, along the x- and y-axis, where a rectangular graphical object is rendered relative to its position. Adjustment is specified for each axis by a real value in the range [0.0, 1.0].

Suppose, for example, that a computer game program renders a text string to the display (320*200 pixels) representing the current game score. Figure 16 illustrates a few different possible placements of the label together with suitable adjustments.

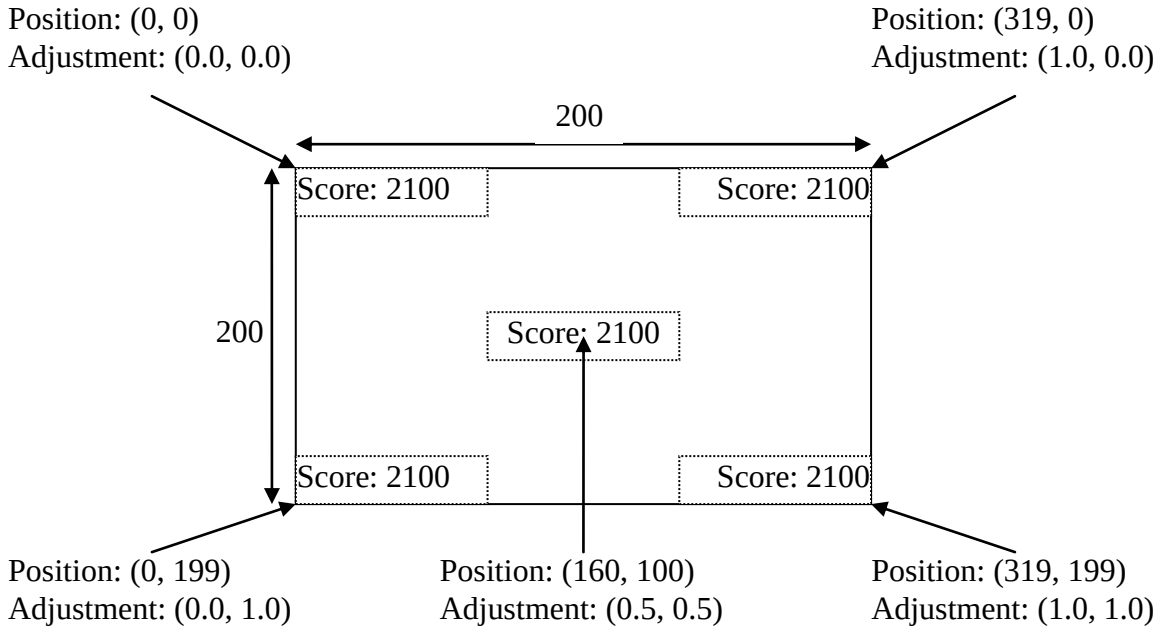


Figure 16. Placement and adjustment.

Note, in the above example, how each position refers to an extreme point on the display (i.e. to one of the corners or to the centre) and that they can continue to do so regardless of the size of the corresponding rectangle, which of course will grow over time as the player gains higher scores.

VGCAdjustment is implemented using the simpler type VGCFraction that represents a real value in the range [0.0, 1.0]:

```
/* Initialization */
VGCFraction fraction0(0.5);
VGCFraction fraction1(fraction1);

/* Assignment */
VGCFraction fraction0(0.5);
VGCFraction fraction1;
fraction1 = fraction0;

/* Reading/Setting */
VGCFraction fraction;                    //Default 0.0
fraction.setFraction(0.5);
```

```
bool b = (0.5 == fraction.getFraction());      //b = true;

/* Comparison */
VGCFraction fraction0(0.5);
VGCFraction fraction1(0.6);
bool b0 = (fraction0 == fraction1);            //b0 = false
bool b1 = (fraction0 != fraction1);            //b1 = true
```

VGCAdjustment, in turn, can be used as follows.

```
/* Initialization*/
VGCAdjustment adjustment0;                //Default; x/y-adjustment = 0.0
VGCAdjustment adjustment1(0.5, 1.0);
VGCAdjustment adjustment2(adjustment1);

/* Assignement */
VGCAdjustment adjustment0(0.1, 0.2);
VGCAdjustment adjustment1;
adjustment1 = adjustment1;

/* Reading/Writing */
VGCAdjustment adjustment;
adjustment.setXAdjustment(0.5);
bool b = (0.5 == adjustment.getXAdjustment().getFraction());     //b = true

/* Comparison */
VGCAdjustment a0(0.1, 0.2);
VGCAdjustment a1(0.2, 0.3);
bool b0 = (a0 == a1);              //b0 = false
bool b1 = (a0 != a1);              //b1 = true
```

3.8 VGCKey

VGCKey defined in the VGCKey component is, as suggested by the name, used to represent individual keys on the keyboard. Each key is represented by its unique key code:

```
NULL_KEY, BACKSPACE_KEY, TAB_KEY, RETURN_KEY, SHIFT_KEY, CONTROL_KEY, MENU_KEY,
ESCAPE_KEY, SPACE_KEY, PAGE_UP_KEY, PAGE_DOWN_KEY, END_KEY, HOME_KEY, ARROW_LEFT_KEY,
ARROW_UP_KEY, ARROW_RIGHT_KEY, ARROW_DOWN_KEY, DELETE_KEY, ZERO_KEY, ONE_KEY,
TWO_KEY, THREE_KEY, FOUR_KEY, FIVE_KEY, SIX_KEY, SEVEN_KEY, EIGHT_KEY, NINE_KEY,
A_KEY, B_KEY, C_KEY, D_KEY, E_KEY, F_KEY, G_KEY, H_KEY, I_KEY, J_KEY, K_KEY, L_KEY,
M_KEY, N_KEY, O_KEY, P_KEY, Q_KEY, R_KEY, S_KEY, T_KEY, U_KEY, V_KEY, W_KEY, X_KEY,
Y_KEY, Z_KEY, PAD_ZERO_KEY, PAD_ONE_KEY, PAD_TWO_KEY, PAD_THREE_KEY, PAD_FOUR_KEY,
PAD_FIVE_KEY, PAD_SIX_KEY, PAD_SEVEN_KEY, PAD_EIGHT_KEY, PAD_NINE_KEY,
PAD_MULTIPLY_KEY, PAD_ADD_KEY, PAD_SUBTRACT_KEY, PAD_DIVIDE_KEY, F1_KEY, F2_KEY,
F3_KEY, F4_KEY, F5_KEY, F6_KEY, F7_KEY, F8_KEY, F9_KEY, F10_KEY, F11_KEY, F12_KEY
```

The null key is used to signify a non existent key, i.e. a key that can never be pressed.

The VGCKey type can be used as follows.

```
/* Initialization */
VGCKey key0;                              //Default, null key
VGCKey escapeKey(VGCKey::ESCAPE_KEY);
VGCKey escapeKey2(escapeKey);

/* Assignment */
VGCKey k0(VGCKey:: A_KEY);
VGCKey k1;
k1 = k0;

/* Reading/Writing */
VGCKey key;
key.setKey(VGCKey::A_KEY);
bool b = (VGCKey::A_KEY == key.getKey());      //b = true
```

```
/* Comparison */
VGCKey k0(VGCKey:: A_KEY);
VGCKey k1(VGCKey:: B_KEY);
bool b0 = (k0 == k1);        //b0 = false
bool b1 = (k0 == k1);        //b1 = true
```

3.9 VGCRandomizer

The VGCRandomizer component implements functions with pseudo random behavior. Before using the component, it must be initialized by a call to the VGCRandomizer::initializeRandomizer() function. When done using it, a client should finalize it by calling the VGCRandomizer::finalizeRandomizer() function. Note that it is perfectly valid for an application to call the initialization function several times. In that case, the component will remain initialized until the finalization function has been called the corresponding number of times.

By using these functions, an application can involve content and/or behavior that vary randomly each time the application is run. For example:

```
VGCRandomizer::initializeRandomizer();

bool criticalHit = VGCRandomizer::getBool(0.1);
if(criticalHit){
     // 10% chance to chop of the opponent's head
}
else{
     // 90% chance to inflict normal damage
}

/* Bullet velocity varies from 200.0 to 250.0 */
double bulletVelocity = VGCRandomizer::getDouble(200.0, 250.0);

/* Bullet mass varies from 1.0f to 2.0f */
float bulletMass = VGCRandomizer::getFloat(1.0f, 2.0f);

/* Rolling a typical dice */
int result = VGCRandomizer::getInt(1, 6);

VGCRandomizer::finalizeRandomizer();
```

3.10 VGCClock

The VGCClock component implements clock and timer functionality. Before using this component it must be initialized by calling VGCClock::initializeClock(). When done using it, it should be finalized by calling VGCClock::finalizeClock(). A client may initialize the component several times. In that case, the component will stay initialized until it has been finalized the corresponding number of times.

3.10.1 Clock functions

The clock measures the time, in seconds, since the component was initialized. For example, it is possible to measure the time it takes to execute a function:

```
const double START_TIME = VGCClock::getTime();
someFunction();
const double END_TIME = VGCClock::getTime();
const double EXECUTION_TIME = (END_TIME – START_TIME);
```

The clock component also implements a wait() function that suspends execution for a specified period of time:

```
VGCClock::wait(0.1);   //Suspend execution for 0.1 seconds
```

3.10.2 Timer functions

Timers are used to measure the time, in seconds, since the timer was created or since it was last reset. Furthermore, each timer is associated with a timeout interval. If this amount of time has passed since the timer was created or since it was last reset, the timer is expired.

Timers are created by calling the VGCClock::openTimer() function. When done using a timer, it should be destroyed by calling the VGCClock::closeTimer() function. To following example illustrates basic usage of timers.

```
/* Create timer*/
const double TIMEOUT = 0.5;                    //Timer expires after 0.5 seconds
VGCTimer timer = VGCClock::openTimer(TIMEOUT);

/* Repeat something for 0.5 seconds */
While(!VGCClock::isExpired(timer)){
     //Something…
}

/* Get time t (0.5 <= t) */
const double CURRENT_TIME = VGCClock::getTime();

/* Increase timeout by 1.5 seconds and wait again */
VGCClock::setTimeout(timer, VGCClock::getTimeout(timer) + 1.5);
VGCClock::reset(timer);
VGCClock::wait(timer); //Suspend execution until timer expires (after 2 seconds)

/* Assignment */
VGCTimer timer2 = timer;     //timer2 refers to the same timer

/* Destroy timer */
VGCClock::closeTimer(timer);
```

3.11 VGCDisplay

The VGCDisplay component implements functions for rendering various graphical objects to the display. Before any rendering can take place, however, a client must initialize the display by calling the VGCDisplay::initializeDisplay() function to specify the application name, display width, and display height. The application name will be displayed on the title bar of the application windows. Width and height refer to the client area of this window and are measured in terms of pixels. For example:

```
 /* Create a display that has a size of 320 by 200 pixels */
const string applicationName = "My application";
const int    DISPLAY_WIDTH   = 320;
const int    DISPLAY_HEIGHT  = 200;
VGCVirtualGameConsole::initialize(applicationName, DISPLAY_WIDTH, DISPLAY_HEIGHT);
```

This results the display window shown in Figure 17.

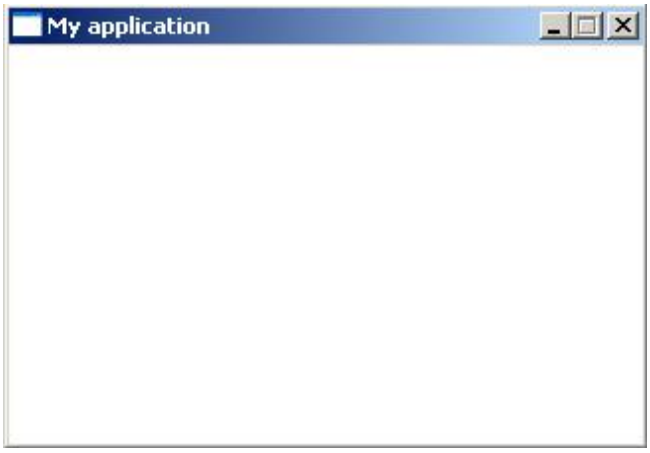


Figure 17. A simple display.

It is valid for an application to call VGCDisplay::initializeDisplay() several times. In that case, VGCDisplay will remain initialized until VGCDisplay::finalizeDisplay() has been called the corresponding number of times. Note, in this case, that only the first call to the initialization function will specify the display window properties. Later calls will simply return without affecting the window.

Before rendering, clients must signal the beginning and ending of each frame by calling VGCDispaly::beginFrame() and VGCDisplay::endFrame() respectively. It is perfectly in order for a client to call the beginFrame() function several times. In order to signal the end of the frame in this case, clients must call the endFrame() function the corresponding number of times.

3.11.1 Clearing the display

The display can be cleared by calling VGCDisplay::clear(). In essence this renders a filled rectangle that fills the entire display area. For example:

```
/* Clear the display area using black color */
VGCColor black(255, 0, 0, 0);
VGCDisplay::clear(black);
```

Note that it is possible to create a blurring/fading effect by using a partially transparent color. For example:

```
/* Fade the display area */
VGCColor black(200, 0, 0, 0);
VGCDisplay::clear(black);
```

(This is however currently a notoriously slow operation on most hardware platforms.)

3.11.2 Basic shapes

VGCDisplay implements the following functions to draw points, lines, rectangles, and ellipses:

```
void renderPoint(
      const VGCColor &color,
      const VGCVector &position
);

void renderLine(
      const VGCColor &color,
      const VGCVector &from,
      const VGCVector &to
);

void renderRectangle(
      const VGCRectangle &rectangle,
      const VGCColor &color,
      const VGCAdjustment &adjustment,
```

```
      bool fill
);

void renderEllipse(
      const VGCRectangle &rectangle,
      const VGCColor &color,
      const VGCAdjustment &adjustment,
      bool fill
);
```

Suppose the display has a size of 320 by 200 pixels and that the rendering of a new frame has been signaled. Then the following code produces the output illustrated in Figure 18.

```
const int DISPLAY_WIDTH  = 320;
const int DISPLAY_HEIGHT = 200;

VGCColor black(255, 0, 0, 0);
VGCColor white(255, 255, 255, 255);

VGCDisplay::clear(white);

/*  Set lower left pixel */
VGCDisplay::renderPoint(
     black,
     VGCVector(0, DISPLAY_HEIGHT - 1)
);

/* Render a line from the upper left to the lower right corner */
VGCDisplay::renderLine(
      black,
      VGCVector(0, 0),
     VGCVector(DISPLAY_WIDTH - 1, DISPLAY_HEIGHT - 1)
);

/* Render 10 by 10 rectangle in the lower right corner */
VGCDisplay::renderRectangle(
     VGCRectangle(VGCVector(DISPLAY_WIDTH - 1, DISPLAY_HEIGHT - 1), 10, 10),
     black,
     VGCAdjustment(1.0, 1.0),
     False
);

/* Render an ellipse that spans the display area */
VGCDisplay::renderEllipse(
     VGCRectangle(VGCVector(0, 0), 320, 200),
     black,
     VGCAdjustment(0.0, 0.0),
     false
);
```

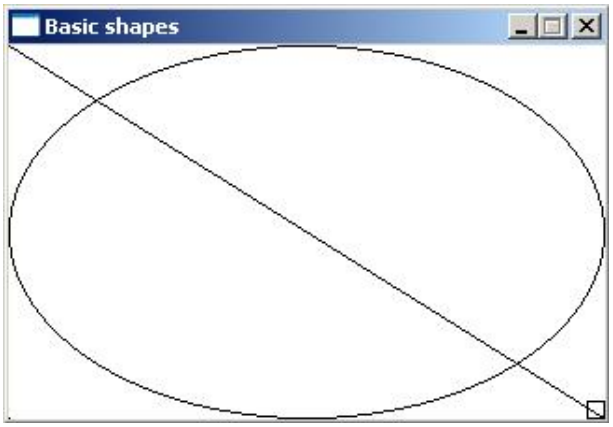


Figure 18. Basic shapes.

Note that rectangles and ellipses with a width or height lesser than two currently will not appear on the screen.

3.11.3 Text

VGCDisplay implements a function for rendering text strings:

```
static void renderString(
      const VGCFont &font,
      const std::string &string,
      const VGCColor &color,
      const VGCVector &position,
      const VGCAdjustment &adjustment
);
```

Before calling this function, however, the client needs to create a font object by calling the VGCDisplay::openFont() function. The parameters for this function specify the font name, e.g. Times New Roman, and size, roughly corresponding to the height of the characters in pixels. When done using a font, it should be closed by calling the VGCDisplay::closeFont() function.

The following program constitutes an example of how to use these functions to output a “Hello World!” on the display. This example was also considered in Chapter 2.

```
#include "VGCVirtualGameConsole.h"
#include <string>

using namespace std;

int VGCMain(const VGCStringVector &arguments){

     const string applicationName = "Hello world";
     const int    DISPLAY_WIDTH   = 320;
     const int    DISPLAY_HEIGHT  = 200;
     VGCVirtualGameConsole::initialize(applicationName, DISPLAY_WIDTH,
DISPLAY_HEIGHT);

     const int FONT_SIZE = 48;
     VGCFont   font      = VGCDisplay::openFont("Times New Roman", FONT_SIZE);

     while(VGCVirtualGameConsole::beginLoop()){

          if(VGCDisplay::beginFrame()){

               const VGCColor backgroundColor = VGCColor(255, 255, 255, 255);
               VGCDisplay::clear(backgroundColor);

               const string        text           = "Hello World!";
               const VGCColor      textColor      = VGCColor(255, 255, 0, 0);
```

```
                const VGCVector     textPosition   = VGCVector(0, 0);
                const VGCAdjustment textAdjustment = VGCAdjustment(0.0, 0.0);
                VGCDisplay::renderString(
                      font,
                      text,
                      textColor,
                      textPosition,
                      textAdjustment);

                VGCDisplay::endFrame();
          }

          VGCVirtualGameConsole::endLoop();
    }

    VGCDisplay::closeFont(font);

    VGCVirtualGameConsole::finalize();

    return 0;
}
```

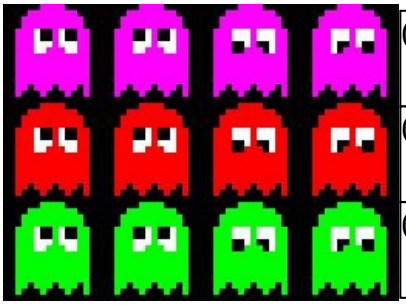
Note that VGCFont has reference semantics. This means that if one font object is assigned to another they actually refer to the same font and that only one of them should be closed eventually.

3.11.4 Images

A simple image, in VGC, consists of pixels arranged into x columns and y rows. The amount of columns is referred to as the width of the image, whereas the number of rows is referred to as the height. Figure 19 shows a simple square image of some water drops.

Figure 19. Drops.

It is also possible, in VGC, to split a simple image into m by n sub images, each of which is referred to as a frame. A specific frame, in this case, is specified by its column and row index respectively. Figure 20 illustrates how this can be used to represent the three ghosts in an imagined pac man clone game using 4 by 3 sub images. The figure also shows the indices for each ghost frame.



(0, 0)	(1, 0)	(2, 0)	(3, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)

Figure 20. Ghost frames.

A simple image, in VGC, is actually a special case of a complex image, i.e. an image with a single frame.

Note, in the case of multiple frames, that the width and height of the image actually refer to the width and height of a frame within that image.

Images are handled by the following functions implemented by VGCDisplay.

```
void     closeImage(const VGCImage &image);
int      getHeight(const VGCImage &image);
int      getWidth(const VGCImage &image);
VGCImage openImage(const std::string &filename, int
void     renderImage(
      const VGCImage &image,
      const VGCVector &frameIndex,
      const VGCVector &position,
      const VGCAdjustment &adjustment
);
```

The following code snippets show how to use these functions. The code is from the Ghost example distributed along with VGC that make the ghosts bounce around the display with psychotic twirling eyes as depicted in Figure 21.

```
/* Open drop image */
const string dropsFilename       = "drops.jpg";
const int    DROPS_X_FRAME_COUNT = 1;
const int    DROPS_Y_FRAME_COUNT = 1;
VGCImage     drops               = VGCDisplay::openImage(
     dropsFilename,
     DROPS_X_FRAME_COUNT,
     DROPS_Y_FRAME_COUNT
);

/* Open ghost image */
const string ghostsFilename        = "ghosts.tif";
const int    GHOSTS_X_FRAME_COUNT  = 4;
const int    GHOSTS_Y_FRAME_COUNT  = 3;
VGCImage ghosts                    = VGCDisplay::openImage(
     ghostsFilename,
     GHOSTS_X_FRAME_COUNT,
     GHOSTS_Y_FRAME_COUNT
);

/* Retrieving the size of a ghost */
const int    GHOST_WIDTH   = VGCDisplay::getWidth(ghosts);
const int    GHOST_HEIGHT  = VGCDisplay::getHeight(ghosts);

/* Render drop image */
const VGCVector     frameIndex = VGCVector(0, 0);
const VGCVector     position   = VGCVector(0, 0);
const VGCAdjustment adjustment = VGCAdjustment(0.0, 0.0);
VGCDisplay::renderImage(drops, frameIndex, position, adjustment);
```

```
/* Closing images */
VGCDisplay::closeImage(ghosts);
VGCDisplay::closeImage(drops);
```

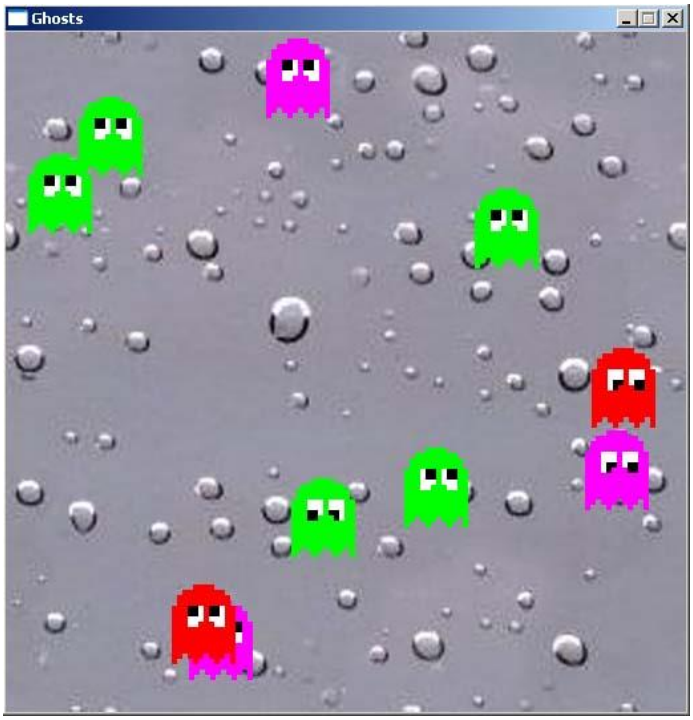


Figure 21. A few ghosts bumping around on the display.

Note that VGCImage has reference semantics. This means that if one image is assigned to another they actually refer to the same image and that only one of them should be closed eventually.

VGC currently supports (to a varying degree) the following image file formats: BMP, GIF, JPEG, PNG, EXIF, WMF, and EMF. In the above example JPEG is used for the background image whereas TIFF is used for the ghosts. TIFF appears to work relatively well when an alpha channel is involved, e.g. to mask away the black pixel in between the ghosts. (A discussion about alpha channels and image formats is beyond the scope of this text, but the reader is encouraged to look it up and/or experiment with graphics programs such as Photoshop or Gimp).

3.11.5 Clipping

It is possible, in VGC, to clip a rendered object by using a clip rectangle. In that case, only the part of the object inside of the rectangle is actually rendered. The clip rectangle is handled using the following functions.

```
const VGCRectangle& getClipRectangle();
void                setClipRectangle(const VGCRectangle &clipRectangle);
```

For example, in order to clip objects to the upper left 25% of the display:

```
const VGCRectangle clipRectangle(
      VGCVector(0, 0),
      DISPLAY_WIDTH / 2,
      DISPLAY_HEIGHT / 2
);
VGCDisplay::setClipRectangle(clipRectangle);
```

Note that no matter how the clip rectangle is placed, objects are also always clipped to fit into the display, i.e. it is not possible to render outside of the display.

3.12 VGCKeyboard

VGCKeyboard implements functions to handle keyboard input:

```
void               beginLoop();
void               clearBuffer();
void               endLoop();
void               finalizeKeyboard();
const std::string& getBuffer();
void               initializeKeyboard();
bool               isPressed(const VGCKey &key);
bool               wasPressed(const VGCKey &key);
```

Before using these functions, the keyboard must be initialized by calling the initializeKeyboard() function. When done using the keyboard it should be finalized using finalizekeyboard(). The keyboard can be initialized several times and will in that case remain initialized until it has been finalized the same number of times.

The beginLoop() and endLoop() functions should be called to signal respectively the beginning and the end of the game loop. (Most clients need not perform this call themselves, since it is called internally by the higher level VGCVirtualGameConsole::beginLoop() function.)

The isPressed() function tells whether or not a particular key on the keyboard is pressed or not. Similarily, wasPressed() tells whether a key has been released which is useful in order to detect key clicks as opposed to key presses.

Character input can be retrived by means of the getBuffer() function that returns the current input buffer in the form of a C++ standard string. A call to the clearBuffer() function clears the input buffer in order and hence prepares it for new input.

The following program allows the user to enter a text string that appears on the display. The example also shows how one can implement a cursor using the character '|'.

```
#include "VGCVirtualGameConsole.h"
#include <string>
#include <strstream>

using namespace std;

int VGCMain(const VGCStringVector &parameters){

     const string applicationName = "Text input";
     const int    DISPLAY_WIDTH   = 640;
     const int    DISPLAY_HEIGHT  = 320;
     VGCVirtualGameConsole::initialize(applicationName, DISPLAY_WIDTH,
DISPLAY_HEIGHT);

     const int FONT_SIZE = 24;
     VGCFont font = VGCDisplay::openFont("Times New Roman", FONT_SIZE);

     while(!VGCKeyboard::wasPressed(VGCKey::ESCAPE_KEY) &&
VGCVirtualGameConsole::beginLoop()){

           if(VGCDisplay::beginFrame()){

                VGCDisplay::clear(VGCColor(255, 255, 255, 255));

                /* Attach cursor | to the character buffer and output it */
                ostrstream output;
                output << VGCKeyboard::getBuffer() << '|' << '\0';
                const VGCColor      color      = VGCColor(255, 255, 0, 0);
                const VGCVector     position   = VGCVector(DISPLAY_WIDTH / 2,
DISPLAY_HEIGHT / 2);
                const VGCAdjustment adjustment = VGCAdjustment(0.5, 0.5);
                VGCDisplay::renderString(font, output.str(), color, position,
adjustment);

                VGCDisplay::endFrame();
```

```
            }

            VGCVirtualGameConsole::endLoop();
      }

      VGCDisplay::closeFont(font);

      VGCVirtualGameConsole::finalize();

      return 0;
}
```

3.13 VGCVirtualGameConsole

The top component in VGC is named VGCVirtualGameConsole and implements only a few fundamental functions for initialization and game loop signaling:

```
bool beginLoop();
void endLoop();
void finalize();
void initialize(const std::string &applicationName, int displayWidth, int
```

The initialize() function initialize all lower level components and may be called any number of times. When done using VGC, a client should finalize VGC by calling finalize() the same number of times.

The beginLoop() and endLoop() functions are called to signal respectively the beginning and ending of the game loop. The beginLoop() function may be called several times as long as endLoop() is called the same number of times. Both these functions also propagate the notification to lower level components that require this, e.g. the VGCkeyboard component.

For convenience, the headerfile (VGCVirtualGameConsole.h) for this component includes all the other public components in the library so a client need never include these as well.

Note that special care is needed to ensure that the game loop actually executes and eventually ends after a call to beginLoop(). This may appear obvious, but consider using the following code to allow a user to exit the program by clicking the escape key.

```
while(VGCVirtualGameConsole::beginLoop()
&& !VGCKeyboard::wasPressed(VGCKey::ESCAPE_KEY)){
      …
}
```

In this case, when escape has been clicked, the beginLoop() function will execute even though the second condition is false causing the program to break out of the loop.

In general, this problem is solved by ensuring that the beginLoop() function is only called when all other loop conditions are true. For the above case, we can achieve this simply by swapping the order of both conditions as follows.

```
while(!VGCKeyboard::wasPressed(VGCKey::ESCAPE_KEY) &&
VGCVirtualGameConsole::beginLoop()){
      …
}
```