

FrontPanelText Sample

*This sample is compatible with the Microsoft Game Development Kit (June 2020)*

# Description

The FrontPanelText sample demonstrates how to use the CPU to draw text on the Xbox One X Devkit and Project Scarlett Devkit Front Panel Displays. The sample uses a class called RasterFont which can load .rasterfont files. As the name suggests, .rasterfont files contain a simple, pixel-based font where each glyph has been rasterized. This format is suitable for rendering on the CPU. The RasterFont class provides printf-style methods that make it easy to render text. For more information on how to create your own .rasterfont files from any True Type font installed on your PC, see also the RasterFontGen sample.

# Building the sample

If using an Xbox One devkit, set the active solution platform to Gaming.Xbox.XboxOne.x64.

If using Project Scarlett, set the active solution platform to Gaming.Xbox.Scarlett.x64.

*For more information, see* Running samples*, in the GDK documentation.*

# Using the sample

The FrontPanelText Sample is intended for the Xbox One X Devkit and the Project Scarlett Devkit with the integrated front panel. When you start the sample, it will render some sample text to the front panel display. Use the front panel DPAD (left, right) to change the font face for the text and the font size (up, down.) Using DPAD up will increase the font size, whereas using DPAD down will decrease the font size.

The DPAD button can also be pressed (select) to capture the buffer from the front panel display and save the result to a .dds file located in the Title Scratch folder.

The following images are screenshots from the sample showing a couple of font options rendered at different sizes:



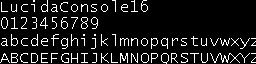
1 LED per Button

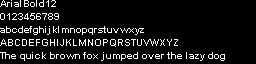
5x Programmable buttons

256 x 64 x 4bpp OLED display

DPAD + Select

Lucida Console is quite readable on the front panel. It is a fixed width font and so is a good choice for menus and widgets since the text will have a predictable layout geometry. Also, it is still readable at a height of 12 pixels and at this size, 5 lines of text fit comfortably on the display.

Here is the same font rendered at a size of 16 pixels. Notice that at this size, the display will accommodate 4 lines of text.

Arial is not fixed-width as you can tell by comparing the width of the lower case alphabet to the upper case alphabet. Arial is readable at a height of 12 pixels and can fit more text horizontally on the display compared to a fixed-width font.

Using the RasterFont toolchain, you can generate .rasterfont files using any TrueType font installed on your system. This is an example of a symbol font. The characters from symbol fonts can be used for rendering simple UI elements (e.g. arrows, buttons, etc.) 

# Implementation notes

Text for the front panel is rendered on the CPU using a RasterFont object. To create a RasterFont object, pass the filename of a .rasterfont file to the constructor. For example:

auto myFont = RasterFont(L"Assets\\LucidaConsole16.rasterfont");

For your own project, you can use the .rasterfont files provided with the sample or you can create your own using the RasterFontGen.exe tool. RasterFontGen will allow you to create .rasterfont files, with various sizes and options, from any True Type font installed on your system.

The following code fragment shows a hypothetical, end-to-end usage of the RasterFont object:

// Load the .rasterfont file

auto myFont = RasterFont(L"Assets\\LucidaConsole16.rasterfont");

// Get the buffer descriptor for the front panel display

BufferDesc fpDesc = m\_frontPanelDisplay->GetBufferDescriptor();

// Draw a formatted string to the buffer

myFont.DrawStringFmt(fpDesc, 0, 0,

L"Simple Addition\n%i + %i = %i",

1, 1, (1 + 1));

// Present the buffer to the front panel

m\_frontPanelDisplay->Present();

BufferDesc is a structure that keeps track of the width and height of a CPU buffer. RasterFont can render text into any address in memory, all it needs is a BufferDesc describing the dimensions of the buffer. To make it easier to target the Front Panel display, the sample uses the FrontPanelDisplay class which manages a buffer for the Front Panel. Use FrontPanelDisplay::GetBufferDescriptor() to get a BufferDesc that is suitable for rendering text to the Front Panel using RasterFont.

DrawStringFmt is used to draw the text to the buffer. This is analogous to the standard library function, printf(). Note that it requires a BufferDesc as well as the x and y coordinates of the text. DrawStringFmt does support line breaks when laying out text.

Here’s a summary of the text rendering methods provided by RasterFont:

// MeastureString and MeasureStringFMt are useful for computing

// text bounds for layout purposes

RECT MeasureString(const wchar\_t \*text) const;

RECT MeasureStringFmt(const wchar\_t \*format, ...) const;

// Basic text rendering to a buffer

void DrawString(const struct BufferDesc &destBuffer, unsigned x, unsigned y,

const wchar\_t \*text) const;

// Formatted text rendering to a buffer

void DrawStringFmt(const struct BufferDesc &destBuffer, unsigned x, unsigned y,

const wchar\_t \*format, ...) const;

// The following DrawString variants provide a shade

// parameter that lets you specify different shades

// of gray

void DrawString(const struct BufferDesc &destBuffer, unsigned x, unsigned y,

uint8\_t shade, const wchar\_t \*text) const;

void DrawStringFmt(const struct BufferDesc &destBuffer, unsigned x, unsigned y,

uint8\_t shade, const wchar\_t \*format, ...) const;

// The glyph-specific methods are used for precisely positioning a single glyph

RECT MeasureGlyph(wchar\_t wch) const;

void DrawGlyph(const struct BufferDesc &destBuffer, unsigned x, unsigned y,

wchar\_t wch, uint8\_t shade = 0xFF) const;

RasterFont Notes:

* The DrawString() variants with the shade parameter can be used for rendering black-on-white text. For example, if you want to use black-on-white text for the selected line in a menu system. Bear in mind that DrawString() doesn’t draw the background pixels, so in order to get black on white text, you will first have to draw a white rectangle. Use the MeasureString methods to determine the bounds of the rectangle you will need.
* MeasureGlyph() and DrawGlyph() are useful for precisely positioning individual glyphs. These methods use only the bounding box for the glyph and don’t consider the spacing between adjacent glyphs and the vertical offset that is used for laying out normal text flow. This allows you to precisely position a glyph. For example, if you are using a glyph from a symbol font as a UI element or widget. (You can find some examples of MeasureGlyph() and DrawGlyph() in the FrontPanelDemo sample.)
* If you are not satisfied with the basic text flow provided by RasterFont, then consider leveraging the underlying RasterGlyphSheet class. This class provides a ForEachGlyph() template that you can use for writing custom text flow implementations. The best examples for how to use ForEachGlyph() can be found in the implementations of the various RasterFont::DrawString\*() methods.

Miscellaneous Implementation Notes:

* On a Xbox One X Devkit or Project Scarlett Devkit, ::XFrontPanelIsAvailable() will return true and the full API will be available. Otherwise, ::XFrontPanelIsAvailable() will return false and other ::XFrontPanel\*() functions will return a failed HRESULT code. (e.g. on an Xbox One, Xbox One S, or any retail console without a physical front panel.)
* It is not necessary to present to the front panel on every frame (::XFrontPanelPresentBuffer()). Instead, you only need to present when one or more pixels has changed. Therefore, the sample has an m\_dirty member that will be set whenever there are changes to the display buffer.
* It is also only necessary to set the light states whenever there are changes.
* You cannot directly access the front panel buffer. Instead, you must manage your own buffer and pass the address of your buffer to ::XFrontPanelPresentBuffer(). Sample::CaptureFrontPanelScreen() simply uses the contents of m\_panelBuffer as the pixel payload for a DDS surface.

# Update history

April 2019, first release of the sample.

November 2019, support for the Project Scarlett Devkit.

# Privacy Statement

When compiling and running a sample, the file name of the sample executable will be sent to Microsoft to help track sample usage. To opt-out of this data collection, you can remove the block of code in Main.cpp labeled “Sample Usage Telemetry”.

For more information about Microsoft’s privacy policies in general, see the [Microsoft Privacy Statement](https://privacy.microsoft.com/en-us/privacystatement/).