VisualIT

Release 2.0.0

Lucide

Mar 20, 2021

VISUALT DOCUMENTATION

CHAPTER

ONE

BASIC CONCEPTS

Here you will learn the terminology and key concepts of VisualT, with as few code references as possible. This is a very recommended read, before diving into any other section.

Tip: If you're familiar with Scratch, you're in luck! VisualT was modelled similarly, graphics-wise.

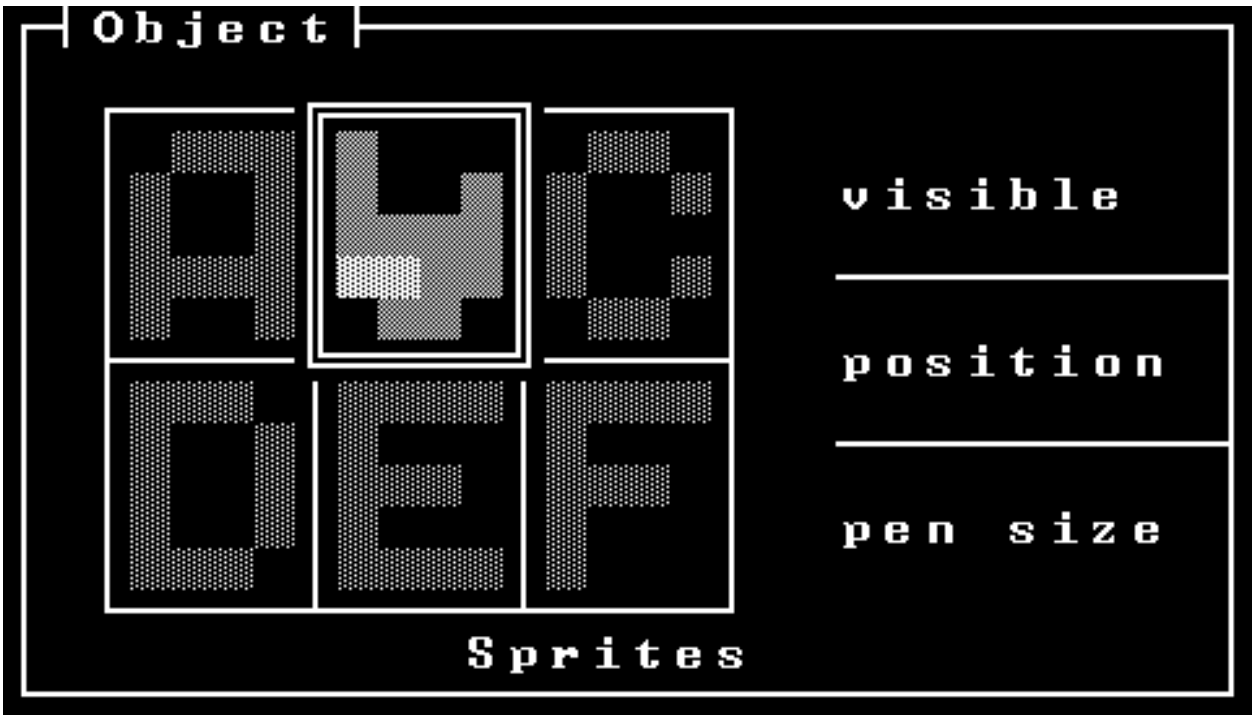
1.1 Objects (obj)

The only top-level graphic element is the **Object**, which contains one or more *Sprites*. An Object can display one and only one Sprite at a time, called the **active Sprite**.

When Objects are passed to a function, their respective active Sprite determines which Sprite will be used. From now on, when we refer to the Sprite of an Object, we mean the active Sprite.

The attributes of an Object are:

- a pointer to the active Sprite
- a visibility flag
- its **x** and **y** position
- its pen size



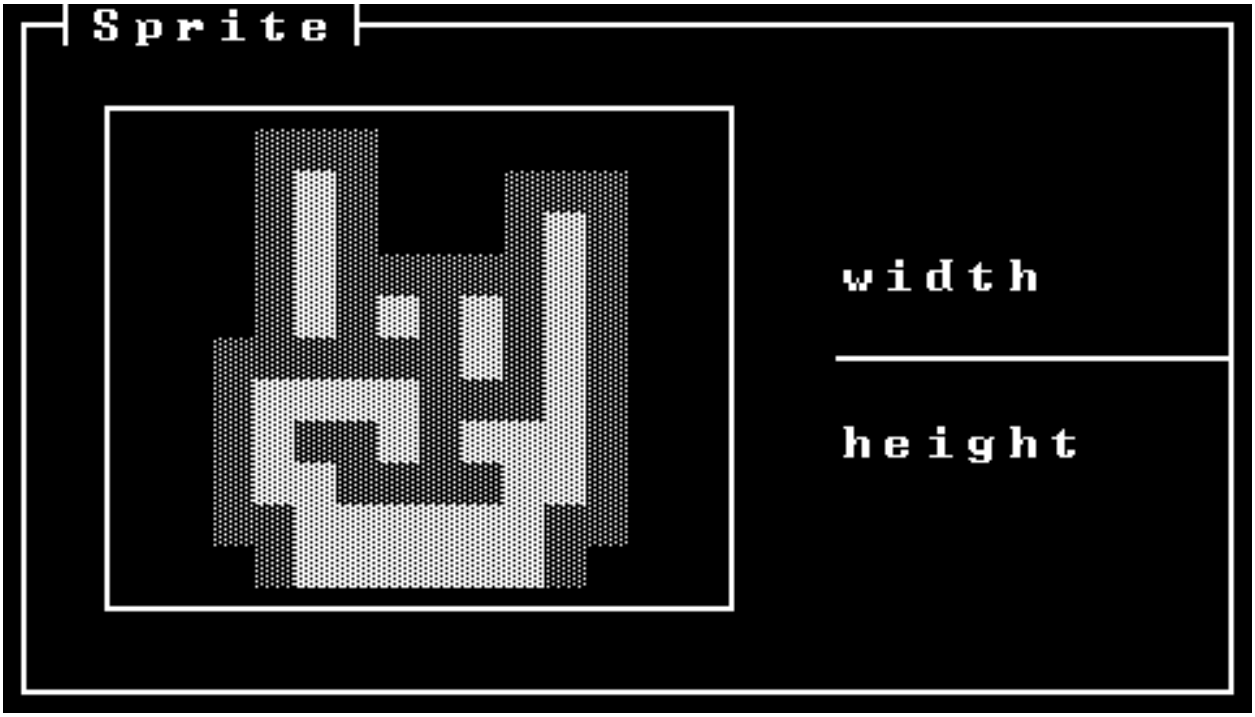
The *visibility flag* indicates if the Object will be drawn (`true`) or skipped (`false`), when passed to drawing functions.

The *pen size* regulates the pen behavior. More on *The Pen*.

Tip: Objects behave similarly to a *Sprite* in Scratch.

1.2 Sprites

Sprites are the text equivalent of a bitmap image, instead of pixels, you have *cells*, where every cell contains a glyph. An Object's active Sprite determines its appearance and size. Sprites are always drawn centered with respect to the Object's position.



Sprites can have transparent cells: a value of 0 (or `\v`, in strings) is interpreted as "no-glyph". When a Sprite is drawn, you'll be able to see the background under transparent areas.

Tip: Sprites are equivalent to a *Costume* in Scratch.

1.3 The Canvas

Objects can be composed and drawn onto another Object that acts as a canvas. So, given two Objects, `a` and `b`, you can either:

- draw `a` on `b`
- draw `b` on `a`

Every Object can act either as a drawable element, or as a drawable surface.

Position [0,0] is placed at the center of the stage, so the maximum values of the **x** and **y** coordinates are respectively half its width and half its height (one half will be one unit greater, if the total is odd).

That does not mean that larger values can't be used: what exceeds the visible area of the canvas simply won't be visible.

Tip: The canvas behaves equivalently to the *Stage* in Scratch.

1.4 The Pen

Objects come equipped with a toggleable *pen* that leaves a mark on the canvas when the Object is moving. It's the classic tool that allows you to draw vector graphics!

You can choose between six stroke sizes:

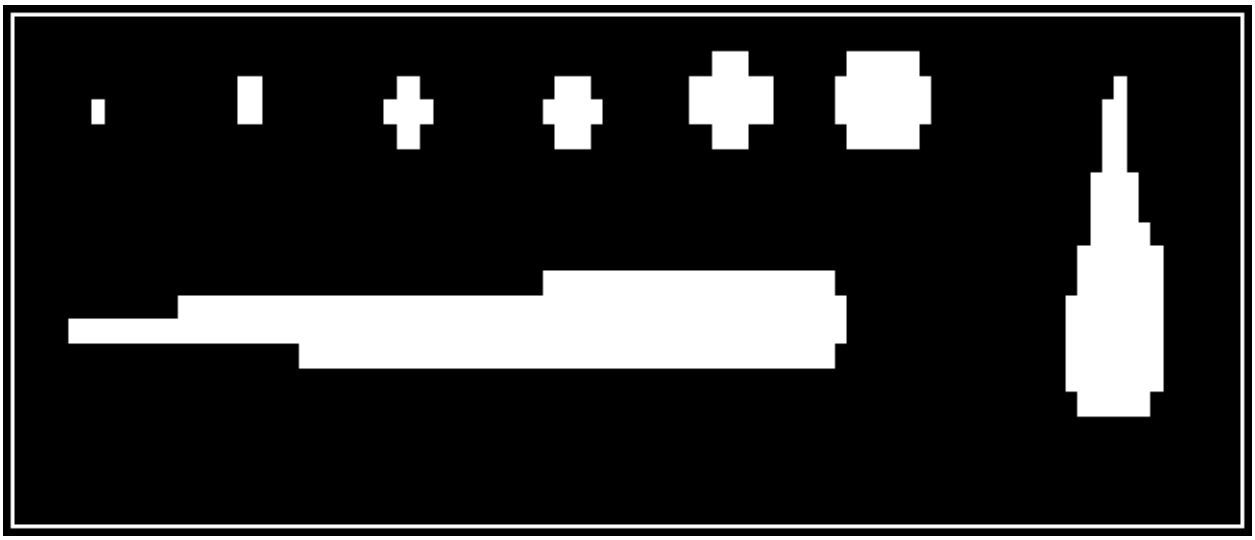


Fig. 1: stroke stencils are wider than tall, to compensate for non-square glyphs

You can use the pen to plot functions, or parametric wireframe stuff. It works even when the drawing Object is hidden.

Fig. 2: the drawing Object is hidden

CHAPTER

TWO

EXAMPLES

Todo: This section is still empty.

CHAPTER

THREE

IMPORT MODULES

Import modules are standalone libraries that provide additional *initializers* capable of parsing third party file types. They can be enabled through the respective `VisualT_<filetype>_IMPORTER` CMake option. Import modules may have third party dependencies.

Ancient versions of VisualT came with their own native Object editor. It was very awkward and inconvenient, so, during the 2.0 library rewrite, I decided to ditch it and adopt an already existing text-art editor. It was a challenging task. The major requirement was non-ascii characters support, which is one of the main features of VisualT.

In the end, I settled on REXPaint, but I wish it wasn't the only one. I'm keeping an eye on lvllvl, for example. If you have suggestions, let me know.

At the moment, the only import module available is the REXPaint import module.

3.1 The REXPaint import module

The REXPaint import module can initialize an Object from one or many .xp files. REXPaint is a popular text-art editor for Windows, available for free on gridsagegames.com. (Should run fine on Wine)

When enabled, an additional zlib dependency will be added to the project. The download and build will be performed automatically by the Hunter package manager. A pre-existing distribution of zlib will probably not be suitable, as the module requires some custom build parameters.

Tip: If you find the plaintext manual a bit hard to read, I'm maintaining an unofficial markdown edition. Contributions welcome!

3.1.1 The REXPaint+VisualT workflow

REXPaint needs some configuration in order to reach an optimum level of productivity. We'll set everything up with the help of the Font Atlas Generator webapp, which I wrote for the purpose.

1. Create a new font atlas

- Choose a reference font to design your Sprites. You could use the default font of your System's terminal, or a readily available one. The generator uses *DejaVu Sans Mono* as default, which is the default font of the Ubuntu terminal. The only requirement is that it must be a **monospaced font**.
- On the generator app, define the **charset** (*glyph palette*) you want to use. Avoid making the character set too large or it'll become awkward to use. The font atlas must be **16 columns wide**. The generator creates square cells by default. While that can be appropriate for bitmap artworks, you'll notice that everything looks squashed horizontally once exported to actual text (just see what happens to the VTCat in the examples!). That's because most font's glyphs are **taller than wide**. Try to find a compromise between fidelity and usability.
- Click on the *save image* and *export charset for REXPaint* buttons to save the output.

2. Add the font to REXPaint (also explained in the REXPaint manual)

- Move to REXPaint's root dir.
- Move the two downloaded files to `data\fonts`.
- Open `_config.xt` with a text editor, the file contains a two-sided text table: `GUI` on the left and `Art` on the right. We'll operate on the `Art` **side**. Add a new row by copy-pasting the `CP437 16x16` line, and change the `Set Name` field as desired. Replace the contents of the `File` field with your atlas' file name, and update the `Columns` and `Rows` values.
- If you want, you can also replace the `Unicode` field with the file name of your charset file. Since REXPaint is a bitmap editor, it does not store any associations between font tiles and their corresponding character, not even in the .xp files. You can provide that data yourself by specifying a *charset file*. That's also the reason why VisualT requires a charset file to import .xp files.

3.1.2 CMake specs

conditions	*VisualT_XP_IMPORTER*
targets	VisualT_xp_importer
components	VisualT_xp_importer
dependencies	VisualT_runtime

See more on *Build instructions*.

3.1.3 API

enum VTXpLoadMode

This enumerator is meant to be used with *vtInitializeXp()* .

.xp files are made up of *n* layers of equal size, which together form an image. This structure is similar to the one used by Objects, where each layer corresponds to a Sprite. The only difference is that Sprites can be of different sizes, while layers cannot. This is why the module can work in two modes:

Values:

enumerator VT_XP_FILES

One Sprite per file: each Sprite is constructed from a different .xp file. This mode allows you to use layers normally in REXPaint, and also to have differently sized Sprites. The downside is that it requires one .xp file per Sprite.

enumerator VT_XP_LAYERS

One Sprite per layer: each Sprite is constructed from a different layer of the same .xp file. This mode allows you to use a single .xp file. The downside is that editing in REXPaint can be less convenient and all Sprites will be the same size.

int **vtInitializeXp**(*VTObj* **obj*, *VTXpLoadMode* *loadMode*, FILE ***restrict** *mapFile*, unsigned int *filesLength*, FILE ***restrict const** **files*)

Initializes `*obj` from one or more .xp files.

.xp files are created by the REXPaint text-art editor. You may pass one or more files depending on `loadMode`. You also need to pass a *charset file* `mapFile`.

Return A value < 0 if the operation failed. An attempt is always made to close `mapFile` and all `files`, even in case of failure, where `*obj` is left uninitialized.

Parameters

- `obj`: a pointer the Object to initialize
- `loadMode`: selects the conversion mode
- `mapFile`: a pointer to the *charset file* that will be used to map tiles indices to their corresponding character
- `filesLength`: the length of `files`. In `VT_XP_LAYERS` mode must be 1.
- `files`: an array of pointers to .xp files

CHAPTER

FOUR

BUILD INSTRUCTIONS

4.1 How do you use a CMake library?

VisualT is a CMake-first library, so it natively supports the most common ways of adding a CMake dependency to a project. If you are just a casual CMake user, this page will offer you some guidance, but if you're completely new, reading further will likely lead to confusion and frustration. In the examples we'll build both the library and the .xp importer.

Tip: If you're interested in learning CMake in an humane way, I can't recommend Craig Scott's Professional CMake: A Practical Guide enough. It's a tad expensive, but for a good reason: once bought, you'll get every future edition for free! That's a very important feature, since CMake updates frequently. You can also hang around the CMake discourse forum.

4.1.1 Approach 1: install locally

VisualT can be installed on your system and automatically found by CMake. Open a terminal inside the project root directory and follow these steps (valid for most CMake projects):

1. **Configure:**

   Here you'll be able to change VisualT's options, if required. Remember to specify the type to avoid unexpected behaviors. On Windows, you may need to run the command within a developer command prompt for certain generators. You can find a list of available options here.

   ```
   cmake -D VisualT_XP_IMPORTER:BOOL=YES -S . -B ./build
   ```

2. **Build:**

   Eventually, you can list the targets you want to build in the `--target` option. You can find a list of available targets here.

   ```
   cmake --build ./build
   ```

3. **Install:**

   Installed files will be listed on the terminal. You can install only components whose targets have been built. You can find a list of available components here.

   ```
   cmake --install ./build
   ```

4. **Use**:

   If you didn't change the install destination, the library will be found automatically:

```
cmake_minimum_required(VERSION 3.17)

project(myProject
        LANGUAGES C)

set(CMAKE_C_STANDARD 99)
set(CMAKE_C_STANDARD_REQUIRED YES)

find_package(VisualT
             REQUIRED
             COMPONENTS
             VisualT_development
             VisualT_import_xp)

add_executable(myCliApp "source.c")
target_link_libraries(myCliApp PRIVATE
                      VisualT::library
                      VisualT::import_xp)
```

4.1.2 Approach 2: using the Hunter package manager

Todo: After the first stable release

4.1.3 Approach 3: include the project as a subdirectory

This is probably the most straightforward way to add local dependencies to a CMake project, but it's rarely a viable approach, as it requires **all** target names to be unique across the whole project. In VisualT, every internal target is prefixed (*namespaced*) with `VisualT_`, and every exported target with `VisualT::`. On top of that, for every linkable target, there's an alias named as its exported name (e.g. `VisualT::library` alias for `VisualT_library`). **The user should link to target aliases only**: that's a good CMake practice that guarantees a certain level of compatibility when switching between internal and exported targets.

```
cmake_minimum_required(VERSION 3.17)

project(myProject
        LANGUAGES C)

set(CMAKE_C_STANDARD 99)
set(CMAKE_C_STANDARD_REQUIRED YES)

# set visualt options (facoltative)
option(VisualT_XP_IMPORTER "Enable the .xp importer library." YES)

add_subdirectory(VisualT)

add_executable(myCliApp "source.c")
target_link_libraries(myCliApp PRIVATE
                      VisualT::library
                      VisualT::import_xp)
```

Add automatic download

You can use the built-in FetchContent CMake module to automate the "download and add as subdirectory" process. The download is performed at configure time. Replace the previous `add_subdirectory(VisualT)` with:

```
include(FetchContent)

fetchcontent_declare(VisualT
                     GIT_REPOSITORY https://github.com/Lucide/VisualT.git
                     GIT_TAG v2.0.0-b)
FetchContent_MakeAvailable(VisualT)
```

4.1.4 Approach 4: external project

ExternalProject is the older method to add dependencies to a project. It's much more generic and versatile than the previous approaches, but this makes it rather difficult to use, with a lot of gotchas. Anyone confident enough with CMake to use this approach needs no further explanation.

4.2 CMake specs

default build type	Release
supported build types	Release, Debug, RelWithDebInfo, MinSizeRel
ALL targets	VisualT_library, VisualT_import_xp[1]
default installation	VisualT::library, VisualT::import_xp[1]

Note: Hunter packages are built at configure time. When using a multi-config generator, they will default to Release.

4.3 Options

VisualT_BUILD_SHARED_LIBS type: `BOOL` default: `NO` Build all libraries (VisualT and modules) as shared libraries. A more specific alternative to *BUILD_SHARED_LIBS*, otherwise completely equivalent.

VisualT_ASSUME_LITTLE_ENDIAN type: `BOOL` default `NO` Skip CMake endianness detection (which can take a bit long), and assume a little endian system, like all x86/x86-64 cpus.

VisualT_PYTHON_VENV type: `PATH` default: `""` Path to a python venv where Sphinx is installed. If your python packages are installed under an unconventional path, use this option to let CMake find them.

VisualT_XP_IMPORTER type: `BOOL` default: `NO` Enable the .xp file import module, and adds a zlib dependency that will be downloaded through Hunter.

[1] If the respective conditions are met.

Common CMake options

BUILD_SHARED_LIBS The standard way to select the type of libraries.

CMAKE_POSITION_INDEPENDENT_CODE This is useful when you want to link a static VisualT to a shared library (instead of an executable). When building static libraries, the export of symbols is automatically disabled, to prevent them from "bleeding out" when linked to shared libraries.

4.4 Targets

A list of all targets that can be defined by VisualT. If specified, the target is defined only if the respective *condition* is met.

target	conditions	description
VisualT_library	none	The VisualT library.
VisualT_tests	none	Some visual feedback tests for VisualT. Currently not very indicative.
VisualT_xp_importer	*VisualT_XP_IMPORTER*	The .xp file import module.
VisualT_doxygen_generate_xml	root project, doxygen found	Step 1/2 for generating documentation.
VisualT_sphinx_generate_html	root project, doxygen found, sphinx found	Step 2/2 for generating documentation.
VTExamples_demo	root project	An example showcasing a variety of VisualT's features, starring the VTCat.
VTExamples_car	root project	An example of a proof of concept game with non-blocking input.
VTExamples_waves	root project	An example showcasing the pen feature.

4.5 Install Components

A list of all install components that can be defined by VisualT. If specified, the component is defined only if the respective *condition* is met. Each component is paired with an export set with the same name. Component dependencies are checked by the config package file.

component	targets	dependencies	conditions	description
VisualT_runtime	VisualT_library	none	none	The VisualT library binaries.
VisualT_development	VisualT_library	VisualT_runtime	none	The VisualT library public header.
VisualT_xp_importer	VisualT_xp_importer	VisualT_runtime	*VisualT_XP_IMPORTER*	The .xp file importer module binaries and public headers.
VisualT_documentation	VisualT_doxygen_generate_xml, VisualT_sphinx_generate_html	none	root project, sphinx found	The html documentation.
VisualT_examples	VTExamples_demo, VTExamples_car, VTExamples_waves	VisualT_development	root project	The examples, as a standalone CMake project.

CHAPTER

FIVE

API REFERENCE

Before reading further, make sure you've understood the *Basic Concepts*.

5.1 Types

5.1.1 Data Types

VisualT's public types are all prefixed with "VT", to avoid potential collision.

typedef uint32_t **VTChar**
: Represents a UTF8-encoded read-only codepoint.

  Being a fully UTF8 compatible library, VisualT stores UTF8 encoded characters in 4 Bytes `uint32_t` variables.

  This is what the API expects when passing a single UTF8-encoded codepoint. Like in `vtFill()` .

typedef struct *VTObj* **VTObj**
: Represents a VisualT Object (*it's not a pointer! this is the real deal*).

  As you can see in the header file, the VTObj `struct` is defined in the public header. *Why is that?* C doesn't have a proper information hiding system, with *opaque pointers* being the only means of hiding the data structure implementation. The trade-off is that the user loses the ability to define statically allocated Objects, and all memory management becomes inaccessible.

  With the "in plain sight" approach, instead, the user can do whatever he wants with memory, as long as he fulfills the unspoken promise to ignore the implementation and work exclusively via API.

5.1.2 Input types

The following types are designed to make the API parameters consistent and easily recognizable. They come already equipped with `const` qualifiers, so they may not necessarily be the best option for representing data in your program, if you plan on doing more advanced manipulations. The only requirement is that the data structures are compatible, so that casts become essentially equivalent to "const cast" (pay attention to type compatibility, if you care about strict aliasing).

typedef uint8_t **const** ***VTStr**
: A pointer to a UTF8-encoded read-only string.

  It's defined this way to enforce the requirement that `char` arrays (strings) must be equivalent to byte arrays.

  This is what the API expects when passing an UTF8-encoded string. Like in `vtSetText()` .

typedef *VTStr* **const** ***VTStrs**
A pointer to a read-only *VTStr* .

This is what the API expects when passing an array of UTF8-encoded strings. Like in *vtInitializeStrings()* .

typedef int **const** (***VTSizes**)[2]
A pointer to a read-only pair of positive integers.

This is what the API expects when passing an array containing multiple Sprite sizes. Like in *vtInitializeBlank()* .

typedef *VTObj* **const** ***const** ***VTObjs**
A pointer to a read-only pointer to read-only *VTObj* .

This is what the API expects when passing an array of pointers to Objects. Like in *vtRender()* .

enum VTDirection
This enumerator is meant to be used whenever the API expects a direction relative to a Sprite.

Sometimes directions can be combined with the bitwise or operator, e.g. `VT_TOP | VT_LEFT`

See *vtAlign()*

Values:

enumerator VTLEFT

enumerator VTRIGHT

enumerator VTTOP

enumerator VTBOTTOM

enum VTSetTextMode
This enumerator is meant to be used with *vtSetText()* .

Values:

enumerator VTCROP

enumerator VTFIT

VTMV
This macro is a more semantic alternative to `NULL` in movement functions.

It resembles “MoVe only”, without being longer than the `NULL` keyword itself.

5.1.3 Literal Helper Macros

You can use the following macros to express **literals** more easily. They are not meant to cast already existing data; use normal casts if you need to do that.

When a literal helper macro can be used on a parameter, it will be indicated in parentheses. See the linked examples.

LTSTR
A helper macro to cast a **literal** string to *VTStr* .

See *vtSetText()*

LTSTRS
A helper macro to cast a **literal** array of strings to *VTStrs* .

The main use case is when you want to initialize an Object from one or more literal strings.

See *vtInitializeStrings()*

LTSIZES
A helper macro to cast a **literal** array of integer pairs to *VTSizes* .

The main use case is when you want to initialize a blank Object given its sprites sizes.

See *vtInitializeBlank()*

LTOBJS
A helper macro to cast a **literal** array of pointers to Objects to *VTObjs* .

An example is when you want to render a group of Objects.

See *vtRender()*

Note: There's no macro to create a *VTChar* literally, because that would break the strict aliasing rule. You can use *vtChar()* instead.

5.2 Functions

As with types, function names are prefixed with "vt", to avoid collisions.

Note:

- Unless explicitly stated (e.g. positioning functions), `NULL` is not an expected input.
- No function expects uninitialized or dangling pointers as input.
- Sizes and lengths must always be greater than 0.

5.2.1 Miscellaneous

void **vtAbout**()
Prints info about VisualT.

Prints the *version*, *build date*, and *GitHub website*.

VTChar **vtChar**(char **const** **ltChar*)
A helper function to convert a **single-character** literal string to a *VTChar* .

Return The first character encoded in a *VTChar* , so that VisualT can process it.

Parameters

- `ltChar`: a UTF8-encoded string containing a single character

5.2.2 Initializers, Releasers and Reallocators

Initializer functions initialize Objects. Be careful to not re-initialize Objects without releasing them before, or you'll cause memory leaks.

They are called "initializers" (and not allocators) because the Object structure itself is not allocated, it's just initialized. You are free to define an Object the way you want, like:

```
VTObj obj;
vtInitializeStrings(&obj, 1, LTSTRS{"An Object in the stack"});
```

or:

```
VTObj *obj = malloc(sizeof(VTObj));
vtInitializeStrings(obj, 1, LTSTRS{"An Object in the heap"});
```

Initializers always set the Object's attributes to the following values:

attribute	default value
active Sprite	Sprite index 0
visibility flag	`true`
position	[0,0]
pen size	0

Note: Reallocators, like *initializers*, perform both memory deallocation and allocation to change the structure of an Object. They might seem convenient in several situations, but be aware of the potential performance impact of using them too often (irrelevant if confronted with I/O times).

---

void **vtInitializeBlank**(*VTObj* *obj, unsigned int *sizesLength*, *VTSizes sizes*)

Initializes `*obj` to a blank Object with `sizesLength` Sprites of `sizes` size.

```
vtInitializeBlank(&canvas, 2, LTSIZES{{10, 5},{5, 10}});
```

The example initializes `canvas` as an Object with two blank (transparent) Sprites, of dimensions 10x5 and 5x10 respectively. This initializer is convenient for setting up an empty canvas.

Parameters

- `obj`: a pointer the Object to initialize
- `sizesLength`: the length of `sizes`, equivalent to the number of Sprites
- `sizes`: (*`LTSIZES`*) an array of integer pairs, each pair represent the dimension of the Sprite

void **vtInitializeArray**(*VTObj* *obj, *VTChar* **const** *v)

Initializes `*obj` from the native *serialized Object array* `v`.

VisualT has its own import and export format, consisting of a sequence of *`VTChar`* that can be embedded in the source as a literal array, or *`read from a file`*. Currently, the only way to serialize an Object is via *`vtSerialize()`* , but it's still a quite powerful tool: it lets you leverage the library to programmatically compose complex scenes from simpler Objects, and then "bake" the result to a new, standalone Object.

Exporters for existing text-art editors may be created in the future.

Parameters

- `obj`: a pointer to the Object to initialize
- `v`: a serialized Object array

int **`vtInitializeFile`**(*VTObj* *obj, FILE *file)

Initializes `*obj` from the *native serialized Object file* pointed by `file`.

A *serialized Object file* is a plain text file containing a *serialized Object array*'s values separated by a space.

See *vtInitializeArray()*

Return A value < 0 if the operation failed. An attempt is always made to close `file`, even in case of failure, where `*obj` is left uninitialized.

Parameters

- `obj`: a pointer the Object to initialize
- `file`: a pointer to a serialized Object file

void **`vtInitializeStrings`**(*VTObj* *obj, unsigned int *utf8StringsLength*, *VTStrs utf8Strings*)

Initializes `*obj` from the array of strings `utf8Strings` of length `utf8StringsLength`.

This initializer is handy for creating Objects from strings on the go. Each *VTStr* becomes a Sprite, rows are delimited by *line feeds* `\n`. Each Sprite will be wide enough to accommodate the longest row, and trailing empty cells will be transparent. Normal space characters will be interpreted literally (opaque), but you can use `\v` to indicate an empty cell.

```
vtInitializeStrings(&obj, 3, LTSTRS{"sprite A","sprite\vB\nsecond\vrow"});
```

- *Sprite 1*: width:8 height:1, fully opaque
- *Sprite 2*: width:10 height:2, all empty cells are transparent

Parameters

- `obj`: a pointer to the Object to initialize
- `utf8StringsLength`: the length of `utf8Strings`, equivalent to the number of Sprites
- `utf8Strings`: (*LTSTRS*) an array of UTF8 strings, each string becomes a Sprite

void **`vtInitializeObj`**(*VTObj* ***restrict** *obj*, *VTObj* **const** ***restrict** *src*)

Initializes `*obj` as a clone of `*src`.

This initializer effectively creates a deep clone of an Object, all the data in the heap is copied. It goes without saying that `*obj` must be uninitialized, and `*src` must not, hence the `restrict` qualifier.

Parameters

- `obj`: a pointer to the Object to initialize
- `src`: a pointer to the source Object

void **`vtRelease`**(unsigned int *objsLength*, *VTObj* **const** ***restrict** **const** **objs*)

Frees the dynamic memory of all the Objects referenced in the array `objs`.

Once released, and Object can be re-initialized safely. You might notice that this function doesn't accept a plain `vtObjs` , but a `restrict`ed qualified variant. This to emphasize that the same Object can't be released twice, not even instances.

Parameters

- `objsLength`: the length of `objs`
- `objs`: (*LTOBJS*) an array of pointers to the Objects you want to release

void **`vtCloneSprite`**(*VTObj* **`const`** **dest*, unsigned int *spriteDest*, *VTObj* **`const`** **src*, unsigned int *spriteSrc*)

A *reallocator* that replaces Sprite n°`spriteDest` of `*dest` with a clone of Sprite n°`spriteSrc` of `*src`.

The source and destination Sprites must be different.

Parameters

- `dest`: a pointer to the Object with the Sprite to replace
- `spriteDest`: the to-be-replaced Sprite number
- `src`: a pointer to the Object with the Sprite to clone
- `spriteSrc`: the to-be-cloned Sprite number

void **`vtResize`**(*VTObj *obj*, unsigned int *width*, unsigned int *height*)

A *reallocator* that resizes `*obj`' Sprite to the defined `width` and `height`. The Sprite is *`cleared`* in the process.

This comes handy when you need to resize the canvas.

Parameters

- `obj`: a pointer to the Object whose Sprite you want to resize
- `width`: the new width
- `height`: the new height

5.2.3 Drawing

void **`vtRender`**(*VTObj* **`const`** **canvas*, unsigned int *objsLength*, *VTObjs objs*)

Draws all the Objects referenced in the `objs` array onto the `*canvas`.

This is the main function that renders your scene. The last Object in the array will be in the topmost layer, followed by the others. Hidden Objects will be skipped.

```
vtRender(&canvas, 3, LTOBJS{&objA, &objB, &objC});
```

The main difference from *`vtStamp()`* , is that the canvas is *`cleared`* in the process.

Parameters

- `canvas`: a pointer to the Object that will act as the canvas
- `objsLength`: the length of `objs`
- `objs`: (*LTOBJS*) an array of pointers to the Objects you want to render

void **`vtStamp`**(*VTObj* **`const`** **canvas*, unsigned int *objsLength*, *VTObjs objs*)

Stamps all the Objects referenced in the `objs` array onto the `*canvas`.

The main difference compared to *`vtRender()`* , is that the pre-existing content on the canvas is left intact, so every Object must be drawn completely, starting from index 0, and eventually overwritten by the following Objects.

Parameters

- `canvas`: a pointer to the Object that will act as the canvas
- `objsLength`: the length of `objs`
- `objs`: (*LTOBJS*) an array of pointers to the Objects you want to stamp

void **vtSetText**(*VTObj* *obj, *VTSetTextMode* mode, *VTStr* utf8String)
Sets `*obj`'s active Sprite content to `utf8String`.

- In `VTFIT` mode, the Sprite will be resized (thus *reallocated*) to fit the content in `utf8String`.
- In `VTCROP` mode, everything that goes outside the active Sprite is instead dropped. The parsing is done the same as in *vtInitializeStrings()* .

```
vtSetText(&obj, LTSTR "A\vKaomoji\n\v(o˘˘o)");
```

Parameters

- `obj`: a pointer to the Object
- `mode`: selects the draw mode
- `utf8String`: (*LTSTR*) an UTF8 string

void **vtClear**(*VTObj* **const** *obj)
Clears `*obj`'s active Sprite. Clearing a Sprite sets all its cell to 0 (transparent).

Equivalent to `fill(&obj, 0);`.

Parameters

- `obj`: a pointer to the Object you want to clear

void **vtFill**(*VTObj* **const** *obj, *VTChar* fillChar)
Sets every cell of `*obj`'s active Sprite to the character in `fillChar`.

```
vtFill(&obj, vtChar(LTSTR ""));
```

Parameters

- `obj`: a pointer to the Object you want to fill
- `fillChar`: (*vtChar()*) the character you want to fill the obj with

void **vtReplace**(*VTObj* **const** *obj, *VTChar* oldChar, *VTChar* newChar)
Replaces every cell of `*obj`'s active Sprite containing `oldChar` with `newChar`.

Parameters

- `obj`: a pointer to the Object
- `oldChar`: (*vtChar()*) the character you want to replace
- `newChar`: (*vtChar()*) the replacing character

void **vtShift**(*VTObj* **const** *obj, *VTDirection* direction, int amount)
Shifts the content of `*obj`'s active Sprite by `amount` cells in the specified `direction`.

Trailing cells will be empty.

Parameters

- `obj`: a pointer to the Object
- `direction`: in which direction to perform the shift
- `amount`: the shift amount. Negative values are supported

void **vtRotate**(*VTObj* **const** **obj*, *VTDirection direction*, int *amount*)
Rotates each row/column of `*obj`'s active Sprite by `amount` cells in the specified `direction`.

Parameters

- `obj`: a pointer to the Object
- `direction`: in which direction to perform the row/column rotation
- `amount`: the rotation amount. Negative values are supported

void **vtOverlay**(*VTObj* **const** **dest*, unsigned int *spriteDest*, *VTObj* **const** **src*, unsigned int *spriteSrc*)
Draws the content of `*src`'s Sprite n°`spriteSrc` onto `*dest`'s Sprite n°`spriteDest` of the same size.

This function exists to solve a very specific task: to flatten the canvas layers. You can set up a canvas with two or more Sprites **of the same size**, and use them as layers (e.g. render layer + pen layer). This function lets you *overlay* a Sprite on another, even if they belong to the same Object.

Parameters

- `dest`: a pointer to the Object with the Sprite on which to draw
- `spriteDest`: the to-be-drawn onto Sprite number
- `src`: a pointer to the Object with the Sprite to overlay
- `spriteSrc`: the to-be-overlaid Sprite number

void **vtDrawAxes**(*VTObj* **const** **obj*)
Draws a visual representation of the coordinate system on `*obj`'s active Sprite.

Parameters

- `obj`: a pointer to the Object

5.2.4 Printers

void **vtPrint**(*VTObj* **const** **obj*, bool *border*)
Prints `*obj` to stdout. If `border` is true, also prints a frame around the output.

The printing is row-buffered: a buffer large enough to store an entire row (a little bit more than 4*width Bytes) is allocated and deallocated by the function. The last character printed is always a *line feed* `\n`.

Parameters

- `obj`: a pointer to the Object you want to print
- `border`: print the frame?

size_t **vtPrintToString**(*VTObj* **const** **obj*, bool *border*, uint8_t ***utf8Buffer*)
Prints `*obj` to `*utf8Buffer`, behaves the same as *`vtPrint()`* . If `*utf8Buffer==NULL`, the buffer will be allocated automatically. There will be no *line feed* `\n` at the end.

- *Usage example 1*: the output buffer is allocated by the function:

```
unsigned char *s = NULL; // remember to initialize the pointer to NULL!
size_t size = printToString(&obj, true, &s); // size = strlen(s) + 1('\0')
puts(s);
free(s); // remember to free the buffer!
```

- *Usage example 2*: the output buffer is allocated by the user:

```
unsigned char *s = malloc(vtPrintStringSize(&obj, true)); // remember to use␣
↪the same border value
size_t size = printToString(&obj, true, &s); // size = strlen(s) + 1('\0')
puts(s);
free(s); // remember to free the buffer!
```

If the provided buffer isn't large enough, the function will cause memory corruption. Use *vtPrintStringSize()* to get the minimum required size.

Return The amount of bytes written by the function, must be to the size of `*utf8Buffer`.

Parameters

- `obj`: a pointer to the Object you want to print
- `border`: print the frame?
- `utf8Buffer`: a pointer to a pointer to a `char` buffer. `*utf8Buffer` will be allocated (thus overwritten) if `NULL`

size_t **vtPrintStringSize**(*VTObj* **const** **obj*, bool *border*)

Calculates the size of the buffer that `*obj` would occupy if printed. With or without `border`.

This is useful if you want to allocate the buffer yourself when using *vtPrintToString()* .

Return The size of the buffer `*obj` would occupy if printed.

Parameters

- `obj`: a pointer to the Object you want to process
- `border`: add the frame to the calculation?

5.2.5 Sprite Operations

unsigned int **vtSprites**(*VTObj* **const** **obj*)

Gets `*obj`'s amount of Sprites.

Return The amount of Sprites.

Parameters

- `obj`: a pointer to the Object

unsigned int **vtSpriteInd**(*VTObj* **const** **obj*)

Gets the index of `*obj`'s active Sprite.

Return The index of the active Sprite.

Parameters

- `obj`: a pointer to the Object

void **vtNextSprite**(*VTObj* *obj)
Sets `*obj`'s next Sprite as active.

Parameters

- `obj`: a pointer to the Object

void **vtPrecSprite**(*VTObj* *obj)
Sets `*obj`'s preceding Sprite as active.

Parameters

- `obj`: a pointer to the Object

void **vtSetSprite**(*VTObj* *obj, unsigned int *index*)
Sets `*obj`'s Sprite of index `index` as active.

Parameters

- `obj`: a pointer to the Object
- `index`: the index of the Sprite you want to set as active

int **vtWidth**(*VTObj* **const** *obj)
Gets the width of `*obj`'s active Sprite.

Return The width of the active Sprite.

Parameters

- `obj`: a pointer to the Object

int **vtHeight**(*VTObj* **const** *obj)
Gets the height of `*obj`'s active Sprite.

Return The height of the active Sprite.

Parameters

- `obj`: a pointer to the Object

int **vtExtremum**(*VTObj* **const** *obj, *VTDirection direction*)
Gets the maximum visible coordinate (extrema) of `*obj`'s active Sprite in the specified `direction`.

Return The maximum visible coordinate of the active Sprite.

Parameters

- `obj`: a pointer to the Object
- `direction`: the axis direction

5.2.6 Object Operations

bool **vtVisible**(*VTObj* **const** *obj)
Gets `*obj`'s visibility flag.

Return `true` if the Object is visible, `false` otherwise.

Parameters

- `obj`: a pointer to the Object

void **vtShow**(*VTObj* *obj)
Sets `*obj`'s visibility flag to `true`.

Parameters

- `obj`: a pointer to the Object

void **vtHide**(*VTObj* *obj)
Sets `*obj`'s visibility flag to `false`.

Parameters

- `obj`: a pointer to the Object

void **vtSetVisibility**(*VTObj* *obj, bool *visible*)
Sets `*obj`'s visibility flag to `visible`.

Parameters

- `obj`: a pointer to the Object
- `visible`: the new value

void **vtSerialize**(*VTObj* **const** *obj, unsigned int *v)
Exports `*obj` as a *serialized object*, and writes it to the array `*v` if `v!=NULL`, otherwise to stdout.

If the provided array isn't large enough, the function will cause memory corruption. Use *vtSerializedArraySize()* to get the minimum required size.

Parameters

- `obj`: a pointer to the processed Object
- `v`: *nullable*: a pointer to a `uint32_t` array

size_t **vtSerializedArraySize**(*VTObj* **const** *obj)
Calculates the size of the array that `*obj` would occupy if serialized.

This is useful if you want to allocate the array yourself when using *vtSerialize()* .

Return The size of the array.

Parameters

- `obj`: a pointer to the processed Object

5.2.7 Pen Operations

VTChar **vtPenChar**(*VTObj* **const** *obj)
Gets `*obj`'s pen character.

Return The pen character of the Object.

Parameters

- `obj`: a pointer to the Object

unsigned short **vtPenSize**(*VTObj* **const** *obj)
Gets `*obj`'s pen size.

Return The pen size of the Object.

Parameters

- `obj`: a pointer to the Object

void **vtSetPenSize**(*VTObj* *obj, unsigned short *size*)
Sets `*obj`'s pen size to `size`.

Parameters

- `obj`: a pointer to the Object
- `size`: the new size, if outside of the [1,6] interval, the pen size defaults to 1

void **`vtSetPenChar`**(*VTObj* **obj*, *VTChar penChar*)
Sets `*obj`'s pen character to `penChar`.

Parameters

- `obj`: a pointer to the Object
- `penChar`: (*`vtChar()`*) the new value

5.2.8 Movement

Movement functions move the Objects in the scene by manipulating their coordinates. They aren't simple setters, because if the `canvas` parameter is not *`VTMV`*, the Object will leave a trail. Read *the pen introduction* to know more.

int **`vtXPosition`**(*VTObj* **`const`** **obj*)
Gets `*obj`'s **x** position.

Return The **x** position of the Object.

Parameters

- `obj`: a pointer to the Object

int **`vtYPosition`**(*VTObj* **`const`** **obj*)
Gets `*obj`'s **y** position.

Return The **y** position of the Object.

Parameters

- `obj`: a pointer to the Object

void **`vtGotoXY`**(*VTObj* **`const`** **canvas*, *VTObj* **obj*, int *x*, int *y*)
Moves `*obj` to the `x`;`y` position. The movement will leave a pen trail on the `*canvas`, if `!=VTMV`.

Parameters

- `canvas`: *nullable*: a pointer to the Object the pen will draw onto, usually the canvas
- `obj`: a pointer to the Object
- `x`: the target **x** coordinate
- `y`: the target **y** coordinate

void **`vtGotoX`**(*VTObj* **`const`** **canvas*, *VTObj* **obj*, int *x*)
Moves `*obj` to `x`. The movement will leave a pen trail on the `*canvas`, if specified.

Parameters

- `canvas`: *nullable*: a pointer to the Object the pen will draw onto, usually the canvas
- `obj`: a pointer to the Object
- `x`: the target **x** coordinate

void **`vtGotoY`**(*VTObj* **`const`** **canvas*, *VTObj* **obj*, int *y*)
Moves `*obj` to `y`. The movement will leave a pen trail on the `*canvas`, if specified.

Parameters

- `canvas`: *nullable*: a pointer to the Object the pen will draw onto, usually the canvas

- `obj`: a pointer to the Object
- `y`: the target **y** coordinate

void **`vtChangeX`**(*VTObj* **`const`** **canvas*, *VTObj* **obj*, int *x*)
Changes `*obj`'s **x** position by `x`. The movement will leave a pen trail on the `*canvas`, if specified.

Parameters

- `canvas`: *nullable*: a pointer to the Object the pen will draw onto, usually the canvas
- `obj`: a pointer to the Object
- `x`: the **x** coordinate shift

void **`vtChangeY`**(*VTObj* **`const`** **canvas*, *VTObj* **obj*, int *y*)
Changes `*obj`'s **y** position by `y`. The movement will leave a pen trail on the `*canvas`, if specified.

Parameters

- `canvas`: *nullable*: a pointer to the Object the pen will draw onto, usually the canvas
- `obj`: a pointer to the Object
- `y`: the **y** coordinate shift

void **`vtAlign`**(*VTObj* **obj*, *VTDirection direction*)
Modifies `*obj`'s coordinates so that the specified `position` matches with the Object position.

Parameters

- `obj`: a pointer to the Object
- `direction`: how to align the Object, see *`VTDirection`*

5.2.9 Sensors

Sensor functions can detect if an Object is touching (overlapping) something else in the scene.

bool **`vtIsTouching`**(*VTObj* **`const`** **canvas*, *VTObj* **`const`** **obj*, unsigned int *objsLength*, *VTObjs objs*)
Detects if `*obj` is overlapping any of the Objects referenced in `objs`.

This sensor detects pixel perfect Object-on-Object collisions, in the space delimited by the `*canvas`. Transparent pixels won't be considered. This is the slowest sensor, as it also allocates memory.

Return `true` if there is a collision, `false` otherwise.

Parameters

- `canvas`: a pointer to the Object whose area will be searched for collisions, usually the canvas
- `obj`: a pointer to the Object you want to process
- `objsLength`: the length of `objs`
- `objs`: (*`LTOBJS`*) an array of pointers to the Objects you want to consider in the test

bool **`vtIsTouchingGlyph`**(*VTObj* **`const`** **canvas*, *VTObj* **`const`** **obj*, *VTChar testChar*)
Detects if `*obj` is overlapping any cell in the `*canvas` containing `testChar`.

This sensor detects pixel perfect Object-on-char collisions. Transparent pixels won't be considered.

Return `true` if there is a collision, `false` otherwise.

Parameters

- `canvas`: a pointer to the Object whose area will be searched for collisions, usually the canvas
- `obj`: a pointer to the Object you want to process
- `testChar`: (`ltChar()`) the character you want to test

bool **`vtIsOutside`** (*VTObj* **`const`** **canvas*, *VTObj* **`const`** **obj*)

Detects if `*obj` is spilling outside the `*canvas` area.

This sensor detects pixel perfect Object-on-area collisions. Transparent pixels won't be considered.

Return `true` if there is a collision, `false` otherwise.

Parameters

- `canvas`: a pointer to the Object whose area will be tested, usually the canvas
- `obj`: a pointer to the Object you want to process

5.3 Implementation Details

> **Warning:** VisualT is currently broken on big endian systems. The issue will be fixed in the following updates.

Restrict qualifier

The `restrict` keyword basically means that the qualified pointer must be unique among the function parameters. In VisualT's API, its meaning is extended to *"the referenced base Object must be unique among the function parameters"*. That's to indicate that multiple pointers to instances created from a common base Object aren't allowed either. This in important to keep in mind when dealing with allocators and releasers, like *`vtRelease()`*. Instances are a more advanced "implicit feature" of VisualT, and should only be used by users familiar with C.

Strict Aliasing

VisualT expects `uint8_t` to be a type compatible with `char`, in order to respect the strict aliasing rule. If, in your implementation, `uint8_t` is not equivalent to `char`, try to disable strict aliasing (-fno-strict-aliasing) and see what happens. If you spot erroneous behavior, create an issue and we'll look into it.

Signedness

Variables in the source default to signed integers. Unsigned integers are used, when possible, to avoid unsigned->signed casts when dealing with unsigned public types. Public types, when reasonable, use unsignedness as a self-explanatory semantic indicator, like `VTObj.length`, which supports `>0` values only. `VTObj.width`, instead, is also a `>0` type, but it's expressed by a signed type, because it's frequent used in calculations involving other signed types like **x** and **y** positions.

CHAPTER

SIX

FREQUENTLY ASKED QUESTIONS

6.1 Why such a high minimum CMake version?

Because CMake is **not** a compiler! In fact, it's designed from the ground up to be updated safely through a sophisticated policies system. Unfortunately, that's a little known fact, and most people stick to ancient version and hurt themselves. Update your CMake!

Tip: If you're using Ubuntu, Kitware provides an APT repository that you can add to your system.

6.2 Oh come on, why no colors?

Colors are cool, but since every terminal does colors differently, VisualT cannot reasonably support them all. If you *need* colors, then NCurses might be a better option. On Windows, things might be even harder.

6.3 Some characters are broken on Windows?

Windows is an ancient OS, a lot of decisions were made before the creation of common standards. The default character encoding is a mixture of UCS-2 and UTF-16. VisualT uses UTF-8, like most of the world today. There's this Stack Overflow answer that explains well the procedure to enable UTF-8 and the potential side effects. Remember to use a unicode capable font!

6.4 Some Objects are not drawn, despite being marked as visible, I checked!

That happens often enough that deserves its own faq entry: check the length parameter in the function calls. Probably it's just C being C .

VisualT is a cross-platform C library that makes creating UTF8 text-based UIs easier and faster.

For a quick overview of the main features, check out the *Basic Concepts* page.

If you appreciated the work, leave a star on the GitHub repo.

Highlights

- Has no dependencies other than standard libraries.
- It works wherever UTF-8 encoded text can be displayed properly, and that includes even Windows 10! See on the *faq*.
- Makes no assumptions about the output, it just outputs text.
- It's *much* more lightweight than more feature-packed libraries like NCurses.
- Good CMake code is one of the main objectives.

Note: The documentation is still work in progress, but the main sections are content complete. That includes:

- *The api documentation*
- *The CMake specs and build instructions*
- *Frequently Asked Questions*

VisualT's dependencies are managed by the Hunter Package Manager:

Hunter is a CMake-powered package manager, it's completely *written* in CMake. Other than being inherently cross-patform, it offers a CMake-friendly interface for every available package, which is a quite valuable feature. It is also committed to complying with modern CMake best practices. If you're starting a new project, give it a try, and spread the word!

INDEX