

Nathan Reed

[Blog](#)[Stuff I've Made](#)[Talks](#)[About Me](#)[← The Many Meanings of "Shader"](#)

A Programmer's Introduction to Unicode

March 3, 2017 · Coding · 20 Comments

Unicode! **U N I C O D E**! **U N I C O D E**! ☹️ The very name strikes fear and awe into the hearts of programmers worldwide. We all know we ought to "support Unicode" in our software (whatever that means—like using `wchar_t` for all the strings, right?). But Unicode can be abstruse, and diving into the thousand-page [Unicode Standard](#) plus its dozens of supplementary [annexes](#), [reports](#), and [notes](#) can be more than a little intimidating. I don't blame programmers for still finding the whole thing mysterious, even 30 years after Unicode's inception.

A few months ago, I got interested in Unicode and decided to spend some time learning more about it in detail. In this article, I'll give an introduction to it from a programmer's point of view.

I'm going to focus on the character set and what's involved in working with strings and files of Unicode text. However, in this article I'm not going to talk about fonts, text layout/shaping/rendering, or localization in detail—those are separate issues, beyond my scope (and knowledge) here.

Contents

Diversity and Inherent Complexity

The Unicode Codespace

Codespace Allocation

Scripts

Usage Frequency

Encodings

UTF-8

UTF-16

Combining Marks

Canonical Equivalence

Normalization Forms

Grapheme Clusters

And More...

Diversity and Inherent Complexity

As soon as you start to study Unicode, it becomes clear that it represents a large jump in complexity over character sets like ASCII that you may be more familiar with. It's not just that Unicode contains a much larger number of characters, although that's part of it. Unicode also has a great deal of internal structure, features, and special cases, making it much more than what one might expect a mere "character set" to be. We'll see some of that later in this article.

When confronting all this complexity, especially as an engineer, it's hard not to find oneself asking, "Why do we need all this? Is this really necessary? Couldn't it be simplified?"

However, Unicode aims to faithfully represent the *entire world's* writing systems. The Unicode Consortium's stated goal is "enabling people around the world to use computers in any language". And as you might imagine, the diversity of written languages is immense! To date, Unicode supports 135 different scripts, covering some 1100 languages, and there's still a long tail of [over 100 unsupported scripts](#), both modern and historical, which people are still working to add.

Given this enormous diversity, it's inevitable that representing it is a complicated project. Unicode embraces that diversity, and accepts the complexity inherent in its mission to include all human writing systems. It doesn't make a lot of trade-offs in the name of simplification, and it makes exceptions to its own rules where necessary to further its mission.

Moreover, Unicode is committed not just to supporting texts in any *single* language, but also to letting multiple languages coexist within one text—which introduces even more complexity.

Most programming languages have libraries available to handle the gory low-level details of text manipulation, but as a programmer, you'll still need to know about certain Unicode features in order to know when and how to apply them. It may take some time to wrap your head around it all, but don't be discouraged—think about the billions of people for whom your software will be more accessible through supporting text in their language. Embrace the complexity!

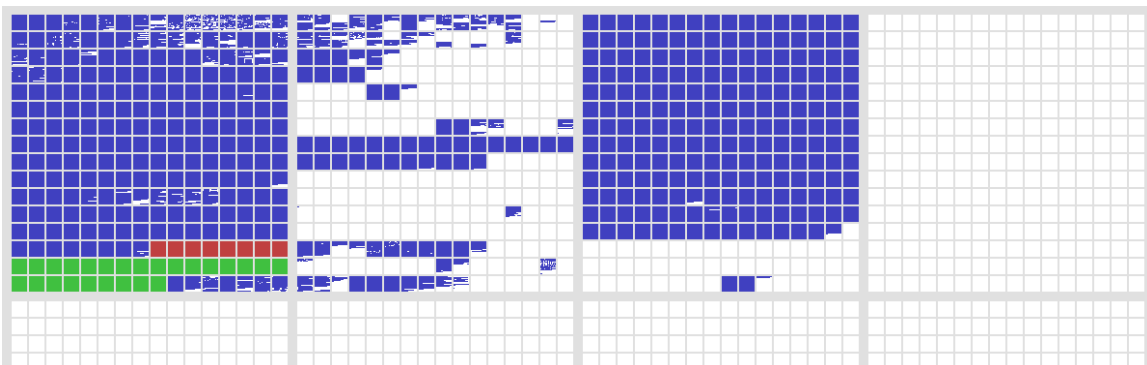
The Unicode Codespace

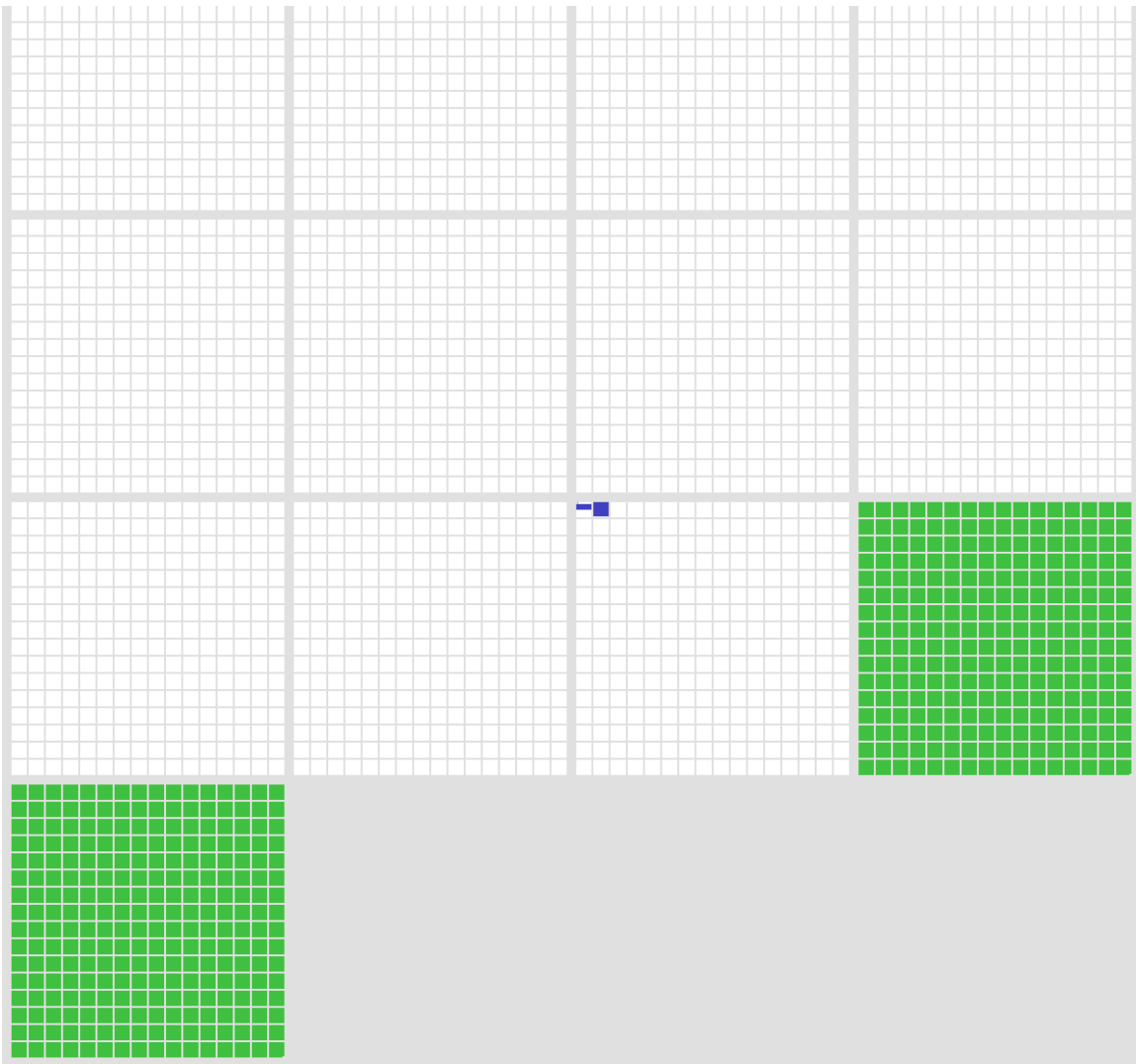
Let's start with some general orientation. The basic elements of Unicode—its “characters”, although that term isn't quite right—are called *code points*. Code points are identified by number, customarily written in hexadecimal with the prefix “U+”, such as [U+0041 “A” LATIN CAPITAL LETTER A](#) or [U+03B8 “θ” GREEK SMALL LETTER THETA](#). Each code point also has a short name, and quite a few other properties, specified in the [Unicode Character Database](#).

The set of all possible code points is called the *codespace*. The Unicode codespace consists of 1,114,112 code points. However, only 128,237 of them—about 12% of the codespace—are actually assigned, to date. There's plenty of room for growth! Unicode also reserves an additional 137,468 code points as “private use” areas, which have no standardized meaning and are available for individual applications to define for their own purposes.

Codespace Allocation

To get a feel for how the codespace is laid out, it's helpful to visualize it. Below is a map of the entire codespace, with one pixel per code point. It's arranged in tiles for visual coherence; each small square is $16 \times 16 = 256$ code points, and each large square is a “plane” of 65,536 code points. There are 17 planes altogether.





White represents unassigned space. Blue is assigned code points, green is private-use areas, and the small red area is surrogates (more about those later). As you can see, the assigned code points are distributed somewhat sparsely, but concentrated in the first three planes.

Plane 0 is also known as the “Basic Multilingual Plane”, or BMP. The BMP contains essentially all the characters needed for modern text in any script, including Latin, Cyrillic, Greek, Han (Chinese), Japanese, Korean, Arabic, Hebrew, Devanagari (Indian), and many more.

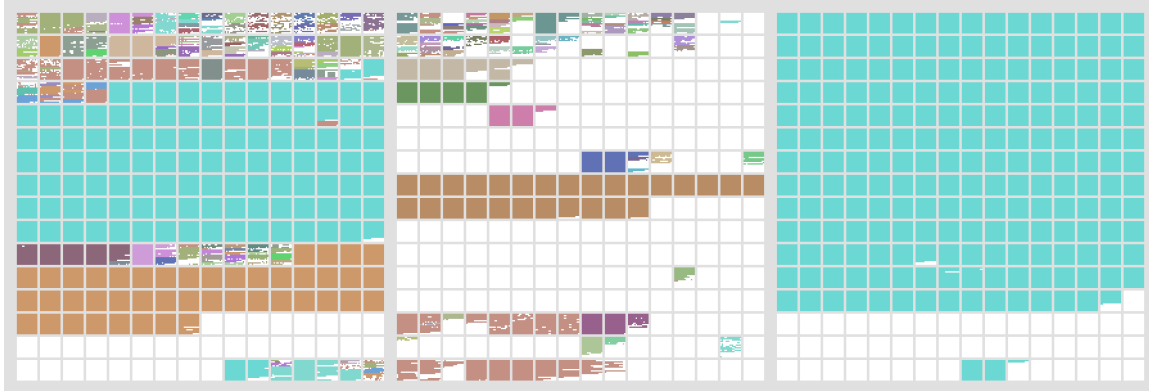
(In the past, the codespace was just the BMP and no more—Unicode was originally conceived as a straightforward 16-bit encoding, with only 65,536 code points. It was expanded to its current size in 1996. However, the vast majority of code points in modern text belong to the BMP.)

Plane 1 contains historical scripts, such as Sumerian cuneiform and Egyptian hieroglyphs, as well as emoji and various other symbols. Plane 2 contains a large

block of less-common and historical Han characters. The remaining planes are empty, except for a small number of rarely-used formatting characters in Plane 14; planes 15–16 are reserved entirely for private use.

Scripts

Let's zoom in on the first three planes, since that's where the action is:

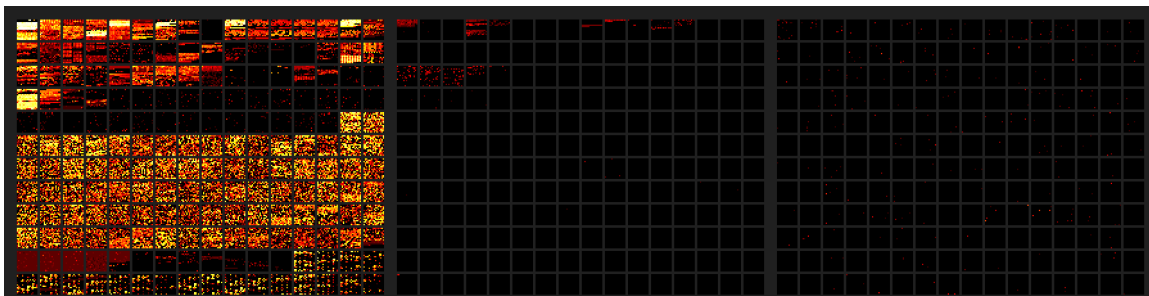


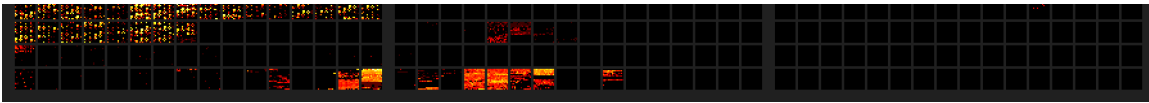
This map color-codes the 135 different scripts in Unicode. You can see how Han () and Korean () take up most of the range of the BMP (the left large square). By contrast, all of the European, Middle Eastern, and South Asian scripts fit into the first row of the BMP in this diagram.

Many areas of the codespace are adapted or copied from earlier encodings. For example, the first 128 code points of Unicode are just a copy of ASCII. This has clear benefits for compatibility—it's easy to losslessly convert texts from smaller encodings into Unicode (and the other direction too, as long as no characters outside the smaller encoding are used).

Usage Frequency

One more interesting way to visualize the codespace is to look at the distribution of usage—in other words, how often each code point is actually used in real-world texts. Below is a heat map of planes 0–2 based on a large sample of text from Wikipedia and Twitter (all languages). Frequency increases from black (never seen) through red and yellow to white.





You can see that the vast majority of this text sample lies in the BMP, with only scattered usage of code points from planes 1–2. The biggest exception is emoji, which show up here as the several bright squares in the bottom row of plane 1.

Encodings

We've seen that Unicode code points are abstractly identified by their index in the codespace, ranging from U+0000 to U+10FFFF. But how do code points get represented as bytes, in memory or in a file?

The most convenient, computer-friendliest (and programmer-friendliest) thing to do would be to just store the code point index as a 32-bit integer. This works, but it consumes 4 bytes per code point, which is sort of a lot. Using 32-bit ints for Unicode will cost you a bunch of extra storage, memory, and performance in bandwidth-bound scenarios, if you work with a lot of text.

Consequently, there are several more-compact encodings for Unicode. The 32-bit integer encoding is officially called UTF-32 (UTF = “Unicode Transformation Format”), but it's rarely used for storage. At most, it comes up sometimes as a temporary internal representation, for examining or operating on the code points in a string.

Much more commonly, you'll see Unicode text encoded as either UTF-8 or UTF-16. These are both *variable-length* encodings, made up of 8-bit or 16-bit units, respectively. In these schemes, code points with smaller index values take up fewer bytes, which saves a lot of memory for typical texts. The trade-off is that processing UTF-8/16 texts is more programmatically involved, and likely slower.

UTF-8

In UTF-8, each code point is stored using 1 to 4 bytes, based on its index value.

UTF-8 uses a system of binary prefixes, in which the high bits of each byte mark whether it's a single byte, the beginning of a multi-byte sequence, or a continuation byte; the remaining bits, concatenated, give the code point index. This table shows how it works:

UTF-8 (binary)	Code point (binary)	Range
0xxxxxxx	xxxxxxx	U+0000–

UTF-8 (binary)	Code point (binary)	Range
		U+007F
110xxxxx 10yyyyyy	xxxxxyyyyyy	U+0080– U+07FF
1110xxxx 10yyyyyy 10zzzzzz	xxxxyyyyyyzzzzzz	U+0800–U+FFFF
11110xxx 10yyyyyy 10zzzzzz 10wwwwww	xxxxyyyyyyzzzzzzwwwwww	U+10000– U+10FFFF

A handy property of UTF-8 is that code points below 128 (ASCII characters) are encoded as single bytes, and all non-ASCII code points are encoded using sequences of bytes 128–255. This has a couple of nice consequences. First, any strings or files out there that are already in ASCII can also be interpreted as UTF-8 without any conversion. Second, lots of widely-used string programming idioms—such as null termination, or delimiters (newlines, tabs, commas, slashes, etc.)—will just work on UTF-8 strings. ASCII bytes never occur inside the encoding of non-ASCII code points, so searching byte-wise for a null terminator or a delimiter will do the right thing.

Thanks to this convenience, it's relatively simple to extend legacy ASCII programs and APIs to handle UTF-8 strings. UTF-8 is very widely used in the Unix/Linux and Web worlds, and many programmers argue [UTF-8 should be the default encoding everywhere](#).

However, UTF-8 isn't a drop-in replacement for ASCII strings in all respects. For instance, code that iterates over the "characters" in a string will need to decode UTF-8 and iterate over code points (or maybe grapheme clusters—more about those later), not bytes. When you measure the "length" of a string, you'll need to think about whether you want the length in bytes, the length in code points, the width of the text when rendered, or something else.

UTF-16

The other encoding that you're likely to encounter is UTF-16. It uses 16-bit words, with each code point stored as either 1 or 2 words.

Like UTF-8, we can express the UTF-16 encoding rules in the form of binary prefixes:

UTF-16 (binary)	Code point (binary)	Range
-----------------	---------------------	-------

UTF-16 (binary)	Code point (binary)	Range
xxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxx	U+0000– U+FFFF
110110xxxxxxxxxxxx 110111yyyyyyyyyyyy	xxxxxxxxxxxxyyyyyyyyyy + 0x10000	U+10000– U+10FFFF

A more common way that people talk about UTF-16 encoding, though, is in terms of code points called “surrogates”. All the code points in the range U+D800–U+DFFF—or in other words, the code points that match the binary prefixes 110110 and 110111 in the table above—are reserved specifically for UTF-16 encoding, and don’t represent any valid characters on their own. They’re only meant to occur in the 2-word encoding pattern above, which is called a “surrogate pair”. Surrogate code points are illegal in any other context! They’re not allowed in UTF-8 or UTF-32 at all.

Historically, UTF-16 is a descendant of the original, pre-1996 versions of Unicode, in which there were only 65,536 code points. The original intention was that there would be no different “encodings”; Unicode was supposed to be a straightforward 16-bit character set. Later, the codespace was expanded to make room for a long tail of less-common (but still important) Han characters, which the Unicode designers didn’t originally plan for. Surrogates were then introduced, as—to put it bluntly—a kludge, allowing 16-bit encodings to access the new code points.

Today, Javascript uses UTF-16 as its standard string representation: if you ask for the length of a string, or iterate over it, etc., the result will be in UTF-16 words, with any code points outside the BMP expressed as surrogate pairs. UTF-16 is also used by the Microsoft Win32 APIs; though Win32 supports either 8-bit or 16-bit strings, the 8-bit version unaccountably still doesn’t support UTF-8—only legacy code-page encodings, like ANSI. This leaves UTF-16 as the only way to get proper Unicode support in Windows.

By the way, UTF-16’s words can be stored either little-endian or big-endian. Unicode has no opinion on that issue, though it does encourage the convention of putting [U+FEFF ZERO WIDTH NO-BREAK SPACE](#) at the top of a UTF-16 file as a [byte-order mark](#), to disambiguate the endianness. (If the file doesn’t match the system’s endianness, the BOM will be decoded as U+FFFE, which isn’t a valid code point.)

Combining Marks

In the story so far, we’ve been focusing on code points. But in Unicode, a

“character” can be more complicated than just an individual code point!

Unicode includes a system for *dynamically composing* characters, by combining multiple code points together. This is used in various ways to gain flexibility without causing a huge combinatorial explosion in the number of code points.

In European languages, for example, this shows up in the application of diacritics to letters. Unicode supports a wide range of diacritics, including acute and grave accents, umlauts, cedillas, and many more. All these diacritics can be applied to any letter of any alphabet—and in fact, *multiple* diacritics can be used on a single letter.

If Unicode tried to assign a distinct code point to every possible combination of letter and diacritics, things would rapidly get out of hand. Instead, the dynamic composition system enables you to construct the character you want, by starting with a base code point (the letter) and appending additional code points, called “combining marks”, to specify the diacritics. When a text renderer sees a sequence like this in a string, it automatically stacks the diacritics over or under the base letter to create a composed character.

For example, the accented character “Á” can be expressed as a string of two code points: U+0041 “A” LATIN CAPITAL LETTER A plus U+0301 “◌̇” COMBINING ACUTE ACCENT. This string automatically gets rendered as a single character: “Á”.

Now, Unicode does also include many “precomposed” code points, each representing a letter with some combination of diacritics already applied, such as U+00C1 “Á” LATIN CAPITAL LETTER A WITH ACUTE or U+1EC7 “ê” LATIN SMALL LETTER E WITH CIRCUMFLEX AND DOT BELOW. I suspect these are mostly inherited from older encodings that were assimilated into Unicode, and kept around for compatibility. In practice, there are precomposed code points for most of the common letter-with-diacritic combinations in European-script languages, so they don't use dynamic composition that much in typical text.

Still, the system of combining marks does allow for an *arbitrary number* of diacritics to be stacked on any base character. The reductio-ad-absurdum of this is [Zalgo text](#), which works by randomly stacking numerous diacritics on each letter, so that they overflow the line spacing and create an appearance of

corruption.

A few other places where dynamic character composition shows up in Unicode:

- [Vowel-pointing notation](#) in Arabic and Hebrew. In these languages, words are normally spelled with some of their vowels left out. They then have diacritic notation to indicate the vowels (used in dictionaries, language-teaching materials, children's books, and such). These

diacritics are expressed with combining marks.

A Hebrew example, with **niqqud**: אַתְּ דַלְתִּי הַיִּזְי הַנִּיעַ, קָטֵב לְשַׁכְּתִי יִשׁוּד
 Normal writing (no niqqud): את דלתי הזיז הניע, קטב לשכתי ישוד

- **Devanagari**, the script used to write Hindi, Sanskrit, and many other South Asian languages, expresses certain vowels as combining marks attached to consonant letters. For example, “ह” + “ि” = “हि” (“h” + “i” = “hi”).
- Korean characters stand for syllables, but they are composed of letters called **jamo** that stand for the vowels and consonants in the syllable. While there are code points for precomposed Korean syllables, it's also possible to dynamically compose them by concatenating their jamo. For example, “ㅎ” + “ㅏ” + “ㄴ” = “한” (“h” + “a” + “n” = “han”).

Canonical Equivalence

In Unicode, precomposed characters exist alongside the dynamic composition system. A consequence of this is that there are multiple ways to express “the same” string—different sequences of code points that result in the same user-perceived characters. For example, as we saw earlier, we can express the character “Á” either as the single code point U+00C1, or as the string of two code points U+0041 U+0301.

Another source of ambiguity is the ordering of multiple diacritics in a single character. Diacritic order matters visually when two diacritics apply to the same side of the base character, e.g. both above: “ā̄” (dot, then macron) is different from “ã” (macron, then dot). However, when diacritics apply to different sides of the character, e.g. one above and one below, then the order doesn't affect rendering. Moreover, a character with multiple diacritics might have one of the diacritics precomposed and others expressed as combining marks.

For example, the Vietnamese letter “ê” can be expressed in *five* different ways:

- Fully precomposed: U+1EC7 “ê”
- Partially precomposed: U+1EB9 “ẹ” + U+0302 “̂”
- Partially precomposed: U+00EA “ê” + U+0323 “̣”
- Fully decomposed: U+0065 “e” + U+0323 “̣” + U+0302 “̂”
- Fully decomposed: U+0065 “e” + U+0302 “̂” + U+0323 “̣”

Unicode refers to set of strings like this as “canonically equivalent”. Canonically equivalent strings are supposed to be treated as identical for purposes of searching, sorting, rendering, text selection, and so on. This has implications for

how you implement operations on text. For example, if an app has a “find in file” operation and the user searches for “ê”, it should, by default, find occurrences of *any* of the five versions of “ê” above!

Normalization Forms

To address the problem of “how to handle canonically equivalent strings”, Unicode defines several *normalization forms*: ways of converting strings into a canonical form so that they can be compared code-point-by-code-point (or byte-by-byte).

The “NFD” normalization form fully *decomposes* every character down to its component base and combining marks, taking apart any precomposed code points in the string. It also sorts the combining marks in each character according to their rendered position, so e.g. diacritics that go below the character come before the ones that go above the character. (It doesn't reorder diacritics in the same rendered position, since their order matters visually, as previously mentioned.)

The “NFC” form, conversely, puts things back together into precomposed code points as much as possible. If an unusual combination of diacritics is called for, there may not be any precomposed code point for it, in which case NFC still precomposes what it can and leaves any remaining combining marks in place (again ordered by rendered position, as in NFD).

There are also forms called NFKD and NFKC. The “K” here refers to *compatibility* decompositions, which cover characters that are “similar” in some sense but not visually identical. However, I'm not going to cover that here.

Grapheme Clusters

As we've seen, Unicode contains various cases where a thing that a user thinks of as a single “character” might actually be made up of multiple code points under the hood. Unicode formalizes this using the notion of a *grapheme cluster*: a string of one or more code points that constitute a single “user-perceived character”.

[UAX #29](#) defines the rules for what, precisely, qualifies as a grapheme cluster. It's approximately “a base code point followed by any number of combining marks”, but the actual definition is a bit more complicated; it accounts for things like Korean jamo, and [emoji ZWJ sequences](#).

The main thing grapheme clusters are used for is text *editing*: they're often the most sensible unit for cursor placement and text selection boundaries. Using

grapheme clusters for these purposes ensures that you can't accidentally chop off some diacritics when you copy-and-paste text, that left/right arrow keys always move the cursor by one visible character, and so on.

Another place where grapheme clusters are useful is in enforcing a string length limit—say, on a database field. While the true, underlying limit might be something like the byte length of the string in UTF-8, you wouldn't want to enforce that by just truncating bytes. At a minimum, you'd want to "round down" to the nearest code point boundary; but even better, round down to the nearest *grapheme cluster boundary*. Otherwise, you might be corrupting the last character by cutting off a diacritic, or interrupting a jamo sequence or ZWJ sequence.

And More...

There's much more that could be said about Unicode from a programmer's perspective! I haven't gotten into such fun topics as case mapping, collation, compatibility decompositions and confusables, Unicode-aware regexes, or bidirectional text. Nor have I said anything yet about implementation issues—how to efficiently store and look-up data about the sparsely-assigned code points, or how to optimize UTF-8 decoding, string comparison, or NFC normalization. Perhaps I'll return to some of those things in future posts.

Unicode is a fascinating and complex system. It has a many-to-one mapping between bytes and code points, and on top of that a many-to-one (or, under some circumstances, many-to-many) mapping between code points and "characters". It has oddball special cases in every corner. But no one ever claimed that representing *all written languages* was going to be easy, and it's clear that we're never going back to the bad old days of a patchwork of incompatible encodings.

Further reading:


- [The Unicode Standard](#)
- [UTF-8 Everywhere Manifesto](#)
- [Dark corners of Unicode](#) by Eevee
- [ICU \(International Components for Unicode\)](#)—C/C++/Java libraries implementing many Unicode algorithms and related things
- [Python 3 Unicode Howto](#)
- [Google Noto Fonts](#)—set of fonts intended to cover all assigned code points

[Tweet](#)

[◀ The Many Meanings of "Shader"](#)

20 Comments on "A Programmer's Introduction to Unicode"

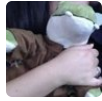
20 Comments reedbeta

 Login ▾ Recommend 8 Share

Sort by Oldest ▾



Join the discussion...

**sdreyesg** • 2 months ago

Man, this was a great post. the explanation of canonical value, the heat map and the unicode codespace chart were my favorite.

7 ^ | ▾ • Reply • Share >

**Corvus** • 2 months ago

I believe «U+0041 "a" latin capital letter a» should read «U+0041 "A" latin capital letter A».

1 ^ | ▾ • Reply • Share >

**Nathan Reed** Author → Corvus • 2 months ago

Thanks; fixed.

1 ^ | ▾ • Reply • Share >

**Pierre Clouthier** • 2 months ago

I've been working with Unicode for years and I've learned many things from this post. Thanks!

3 ^ | ▾ • Reply • Share >

**Aron** • 2 months ago

Really well written and comprehensive post! Thank you so much! Finally I understood Normalization on unicode! haha

1 ^ | ▾ • Reply • Share >

**Kauê Rodrigues** • 2 months ago

Really informative and well written, thank you.

1 ^ | ▾ • Reply • Share >

**Miodrag Milić** • 2 months ago

Thanks a lot... now stuff finally makes sense.

^ | ▾ • Reply • Share >

**Jeff Davies** • 2 months ago

Subscribe



Recent Posts

[A Programmer's Introduction to Unicode](#)

[The Many Meanings of "Shader"](#)

[Tessellation Modes Quick Reference](#)

[little-py-site](#)

[Star Trek: TNG Theme Reorchestration](#)

[EEVEE.WAD Doom Map](#)

[All Posts >](#)

Categories

[Graphics \(20\)](#)

[Coding \(18\)](#)

[Math \(12\)](#)

[GPU \(11\)](#)

[Physics \(6\)](#)

[Eye Candy \(4\)](#)