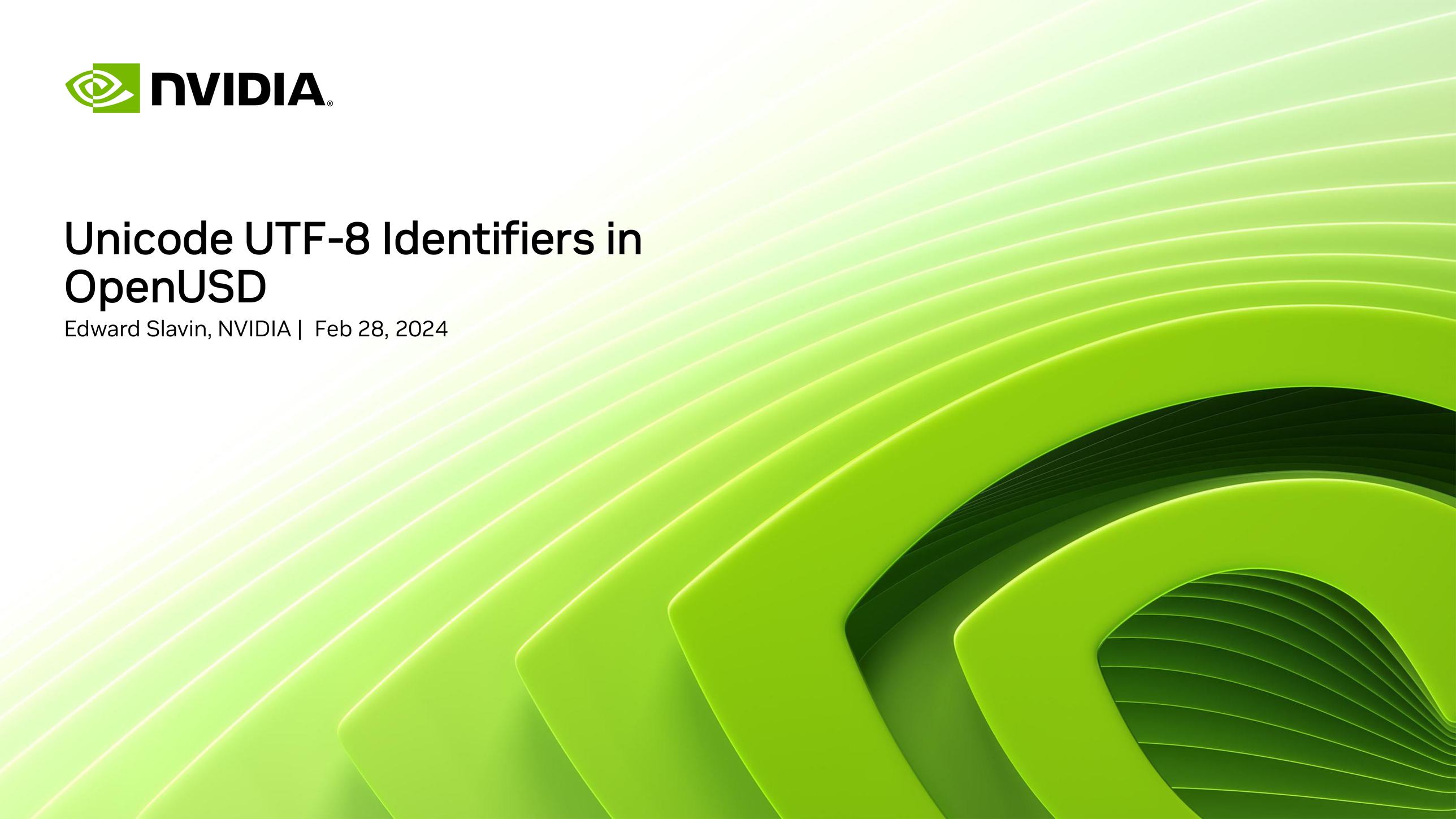

Unicode UTF-8 Identifiers in OpenUSD

Edward Slavin, NVIDIA | Feb 28, 2024

Agenda

- OpenUSD and Identifiers (23.11 vs. 24.02)
- UTF-8 Encoding and Unicode Code Points
- Processing UTF-8 Encoded Strings
- Updating the Path and USDA Parsers
- Special Considerations

Unicode Identifiers in OpenUSD

State of the World

Valid in OpenUSD 23.11

```
#usda 1.0

def “prim” (                                ✓
   customData = {
      string "München" = "hello“            ✓
   }
  _myMetadata = “myValue”                   ✓
)
{
   int myProperty = 3                       ✓
}
```

Invalid in OpenUSD 23.11

```
#usda 1.0

def “München” (                             ✗
   customData = {
      string "München" = "hello“            ✓
   }
    München = “myValue”                     ✗
)
{
   int München = 3                          ✗
}
```

OpenUSD 23.11 Identifiers

ASCII Based

- Identifiers are lexical tokens used to name things
  - Prims
  - Properties
  - Metadata Keys

- Rules for determining if an identifier is valid
  - Largely derived from C++ / Python identifier rules

- Current validity rules are ASCII based
  - Must start with `_` or [A-Za-z]
  - Must continue with `_` or [A-Za-z0-9]

- **Downside**: Can’t use names natively from non-Latin based languages (or even extended Latin-based character sets)

Moving to a World with Unicode Identifiers

Lots of Challenges and Questions

- What is the right string representation? Do we support multiple?
- What sequences of characters should be considered valid identifiers?
  - Could we have paths like /🌍/Forest/🐇/◿?
- What is the surface area of impact?
  - String utilities in Tf
  - String ordering for prim children / properties
  - Path interpretation (parsing, validity, etc.)
  - Text format ingestion (USDA parsing)
  - Additional validation methods across core data model
- The solution must work for the existing OpenUSD Ecosystem
- Should be straightforward for developers to work with
- Need minimal impact to performance, especially for common operations
- Try not to introduce additional external dependencies

Character Representation

Unicode and UTF-8

- Unicode is a specification that assigns a unique *code point* to every character used in human language
- An *encoding* dictates how to represent and interpret that code point to create a contract for interchange
  - Lots of different encodings (ASCII, UCS-2, UCS-4, UTF-8, UTF-16, UTF-32, etc.)
- Unicode code points are almost always represented in UCS-4 (4 bytes/code point)
- UTF-8 is an alternate variable length encoding that can represent a code point in 1-4 bytes
  - Backward compatible with ASCII

Encoding examples:

Character	UCS-4 Code Point	UTF-8 Encoding
A	U+0041	\x41
ü	U+00FC	\xC3 \xBC
∫	U+222B	\xE2 \x88 \xAB
□ (padding character)	U+0080	\xC2 \x80
𫝁	U+2B741	\xF0 \xAB \x9D \x81
😀	U+1F600	\xF0 \x9F \x98 \x80

Decision: Select UTF-8 as the Standard Character Encoding

- Why UTF-8?
  - Defacto standard used everywhere [UTF-8 Everywhere]
  - Efficient encoding – common ASCII text still takes 1 byte / character
  - Backward compatibility – ASCII values are the same
  - No special support needed in C++ / Python
    - std::string is text represented as a sequence of bytes and can be interpreted as UTF-8 encoded characters
    - Python strings are natively Unicode and UTF-8 is the default encoding for source files
- What implications does the encoding choice have for Unicode character processing?
  - Unicode reasoning is done on code points
  - Need mechanisms to convert UTF-8 encoded characters to their equivalent code point representations
  - Need additional methods to reason on code points
    - What class does a character belong to?
    - How can I iterate the code points in a UTF-8 encoded string?
    - How do I order strings?

Valid Identifiers

The XID Derived Properties Classes

- Unicode defines a set of *character properties* and one of these, *general category,* are assigned to each code point
  - Uppercase letters, numbers, symbols, control characters, spacing mark, punctuation, etc.
- *Derived character properties* specify classes that contain characters from different general categories
- Unicode makes recommendations for what character sequences should be considered as identifiers (TR#31)
  - `XID_Start` (Derived from `ID_Start`)
  - `XID_Continue` (Derived from `ID_Continue`)

`XID_Start`

General_Category of uppercase letters, lowercase letters, titlecase letters, modifier letters, other letters, letter numbers, plus Other_ID_Start, minus Pattern_Syntax and Pattern_White_Space code points plus some additional NFKC-based modifications for certain characters.

`XID_Continue`

Include XID_Start characters, plus in the General_Category of nonspacing marks, spacing combining marks, decimal number, connector punctuation, plus Other_ID_Continue, minus Pattern_Syntax and Pattern_White_Space code points plus some additional NFKC-based modifications for certain characters.

`Examples:`

myIdentifier0	✓
_myIdentifier01	✓
München	✓
𝑖75_hgòð쬠	✓
23myIdentifier	✗ (Starts with XID_Continue)
‿Identifier	✗ (Starts with XID_Continue)
恸01₈∫	✗ (Contains symbol not in either)

Processing UTF-8 Encoded Strings

Iterating Code Points to Reason About Classes

- To determine validity of an identifier, we need to know:
  - The code point representation of each character
  - Which character class the code point falls into
- Step 1: Represent the character classes themselves
  - Process the DerivedCharacterProperties file contained in the Unicode database
  - Efficiently represent the code points that fall in the character classes XID_Start / XID_Continue
  - Ranges represented as compile –time arrays of pairs of code points
  - Static `TfUnicodeXidStartFlagData` / `TfUnicodeXidContinueFlagData` bitsets used for efficient lookup
- Step 2: Provide a way to iterate a UTF-8 encoded string and retrieve code points for each character
  - `TfUtf8CodePoint` provides a structure representing a 32-bit code point value
  - `TfUtf8CodePointIterator` provides an iterator for taking a `std::stringview` and iterating the content, converting each UTF-8 encoded character to the equivalent code point (also handles invalid code points, surrogates, etc.)
  - `TfUtf8CodePointView` provides a wrapper around `std::stringview` that can be iterated as code points rather than bytes
- Step 3: Provide a way to determine whether a code point exists in either `XID_Start` or `XID_Continue`
  - `TfIsUtf8CodePointXidStart`
  - `TfIsUtf8CodePointXidContinue`

Iterating a UTF-8 String

- Example 1: Using TfUtf8CodePoint
  - Get a code point from a uint32 value
  - Get a code point for an ASCII character
  - Retrieve the uint32 value of a code point
- Example 2: Iterate a stringview using TfUtf8CodePointView
  - Primary use case, used with range based for loops to iterate all code points
- Example 3: Iterate a string directly using TfUtf8CodePointIterator
  - Used when you need granular access to underlying string positions

Example 1:

```
TfStringify(TfUtf8CodePoint(97)) == "a“
TfUtf8CodePointFromAscii('a') == TfUtf8CodePoint(97)
TfUtf8CodePointFromAscii(‘a’).AsUint32() == 97
```

Example 2:

```
const std::string_view s1{"𝑖75_hgòð愨"};
TfUtf8CodePointView u1{s1};
for (const auto codePoint : u1)
{
   std::cout << codePoint << std::endl;
}
```

Example 3:

```
const std::string s1{"𝑖75_hgòð愨"};
TfUtf8CodePointView view {s1};
TfUtf8CodePointIterator iterator = view.begin();
TfUtf8CodePointIterator anchor = iterator;
for(; iterator != view.end(); ++iterator) {
  if (TfIsUtf8CodePointXidContinue(*iterator)) {
    anchor = iterator;
    break;
  }
}

std::string substr = std::string(view.begin().GetBase(),
anchor.GetBase());
```

The Path Parser

Supporting Unicode Identifiers in Prim and Property Paths

- When a `SdfPath` object gets constructed from a string, that string is parsed and each identifier within that path is validated
- Parsing of the string is done according to the OpenUSD Path Grammar [currently under specification in AOUSD]
- Small updates needed to be made to use the new iterators and character class functions to implement new identifier validation rules
- Note: Notice that the ‘/’ and ‘.’ characters are still used in the path to separate prim children and property children. These characters (and others) are reserved in the path grammar and cannot be used in identifiers, even if they fall in `XID_Start` / `XID_Continue`.

/Scope/Materials/MyMaterial.opacity

/範囲/材料/私の資料.不透明度

The USDA Parser

Supporting Unicode Identifiers in Scene Description

- Reading USDA content is handled by the TextFileFormat plugin
- Parses text content using a flex / bison based parser, grammar is much more complex than the path grammar (OpenUSD USDA grammar is also under specification at AOUSD)
- Conceptually, the changes required are isolated to lexing out `TOK_IDENTIFIER` and `TOK_NAMESPACED_IDENTIFIER`
  - Not as straightforward to granularly specify UTF-8 ranges in flex
  - Need a first validity check of the lexed content before passing on to parser
  - A second validity check is then done as part of scene description creation, where appropriate

```
#sdf 1.4.32
(
    defaultPrim = "_Süßigkeiten"
    doc = """Tests UTF-8 content in prim names and custom
data."""
)

def Xform "_Süßigkeiten" (
    customData = {
        int 存在する = 7
    }
)
{
    custom double3 xformOp:translate = (4, 5, 6)
    uniform token[] xformOpOrder = ["xformOp:translate"]
    string utf8_情報 = "._"جيد

    def Sphere "𝑖573"
    {
        float3[] extent = [(-2, -2, -2), (2, 2, 2)]
        double radius = 2
    }
}
```

General Considerations

Some things to think about using Unicode and UTF-8 Encoding

- This is a great reference for things to think about using Unicode in OpenUSD
- String input that goes through these processing and validation methods must be UTF-8 encoded
  - Source code files containing paths
  - USDA file content
  - OpenUSD will assume that all C++ string content is UTF-8 encoded (tokens, scene paths, asset paths, etc.) unless otherwise specified
  - The transition between Python and C++ should be transparent in most cases due to Boost Python converters registered to Tf
- Full string processing utilities left to higher level libraries
  - Don’t panic! Most string methods in `Tf` will still work for most common usages (e.g., split on a single ASCII character)
  - Functionality for case folding, collation, etc. are not provided in OpenUSD
- Normalization
  - OpenUSD does not provide facilities for string normalization
  - Two strings that are semantically equivalent, but not byte equivalent, will not be considered equal (e.g., VII, Ⅶ)
  - This also implies that two paths that are not byte equivalent will not be considered paths to the same prim / property!
  - Recommend normalizing prior to construction and use
- Identifier Validity
  - Use `SdfPath::IsValidIdentifier` or `SdfPath::IsValidNamespacedIdentifier` rather than `TfIsValidIdentifier` and `TfMakeValidIdentifier`

Unicode Identifiers in OpenUSD

Maybe Someday ☺

Valid in OpenUSD 24.02

```
#usda 1.0

def “München” (                          ✓
   customData = {
      string "München" = "hello“         ✓
   }
    München = “myValue”                  ✓
)
{
   int München = 3                       ✓
}
```

Invalid in OpenUSD 24.02

```
#usda 1.0

def “075MyPrim” (                        ✖
   customData = {
      string "München" = "hello“         ✓
   }
    🐕 = “myValue”                       ✖
)
{
   int Mün-chen = 3                      ✖
}
```