



μCSS

Ein Node-basierter CSS- und Web-Grafik-Prozessor.

npm-Paket: `gulp-mu-css`

Handbuch

Version 2.2.3

2026-06-14

Inhalt

1 Einführung.....	4
1.1 Einordnung: Warum µCSS?.....	4
2 Installation und Start.....	6
3 Anwendungsfälle.....	7
3.1 Benannte Eigenschaften (Variablen).....	7
3.2 Berechnungen und Ausdrücke.....	7
3.3 Mehrere Layouts über Skin-Manifeste.....	8
3.4 Automatische Sprite-Generierung.....	8
3.5 Bilder vorladen (Preload).....	9
3.6 Cursor mit Hotspot und Fallback.....	9
3.7 Automatische Bilderzeugung (media-Steps).....	9
3.7.1 Zwischenschritte raw → final ('outputBase').....	10
4 Grundideen von µCSS.....	12
4.1 Das .µ.css-Format.....	12
4.2 Werte-Interpolation µ(Ausdruck).....	12
4.3 Direktiven -µ: Ausdruck.....	12
4.4 Das Skin-Manifest.....	13
4.5 JavaScript ohne Ersatzzeichen.....	13
5 Build-Ablauf und Gulp-Integration.....	14
5.1 Verzeichniskonventionen.....	14
5.2 Kompilierungs-Ablauf.....	14
5.3 Inkrementeller Build und Cache.....	14
5.4 Gulp-Tasks und Watch.....	15
6 Manifest-Referenz (DefineSkin).....	16
6.1 vars.....	16
6.2 helpers.....	16
6.3 cursors.....	16
6.4 media.....	16
6.5 imageFormat.....	17
6.6 sprites.....	18
6.7 files.....	18
7 µ-Auswertungskontext (Referenz).....	19
7.1 Das \$-Objekt.....	19
7.2 Farb- und Wert-Funktionen.....	19
7.3 Regel- und Dokument-Methoden.....	19
7.4 Sprite().....	20
7.5 Cursor().....	20
7.6 Helper-Funktionen und this-Bindung.....	20
7.7 afterWork-Hooks.....	21
8 Arbeiten mit Farben.....	22
8.1 Farben definieren.....	22
8.2 Alpha-Werte.....	22
8.3 Farbberechnungen.....	22
9 Node-API.....	23
10 Fehlerdiagnostik.....	24
11 Migration von µCSS.....	25
12 Versionshistorie.....	26
13 Rechtliches.....	28
13.1 MIT-Lizenz (Originaltext).....	28

13.2 Marken.....	28
13.3 Verwendete Bibliotheken.....	28

1 Einführung

μCSS ist ein Node-Modul zur Verarbeitung von CSS-Dateien und zur Generierung von Web-Grafiken. Dieses Handbuch beschreibt die Version 2 — den Node-basierten Nachfolger des 2013 eingeführten, Adobe-Photoshop-basierten μCSS 1: Aus erweiterten Quell-Stylesheets (`.μ.css`) und den Medienquellen (z. B. PSD-Entwürfen) entstehen per Gulp die fertigen Skin-Dateien — Standard-CSS plus alle benötigten Bilder, Sprites, Cursor, Fonts und Sounds. Die Bilderzeugung übernimmt das Schwester-Modul μPS, das die alten PS-Plugins ohne Adobe-Abhängigkeit nachbildet.

Da das μ-Zeichen in Paket- und Repository-Namen (npm, git) immer wieder Probleme bereitet, lauten die technischen Namen `gulp-mu-css` und `gulp-mu-ps` — μCSS und μPS sind die Anzeigenamen.

Die wichtigsten Eigenschaften von μCSS 2:

- Quell-Stylesheets bleiben **syntaktisch valides CSS** — Editoren, Linter und Diff-Werkzeuge funktionieren unverändert.
- **Benannte CSS-Eigenschaften** über Skin-Variablen (`$.name`).
- **Berechnung von CSS-Werten durch JavaScript-Ausdrücke** direkt im Stylesheet — ohne Ersatzzeichen wie `«»j`.
- **Mehrere Layouts (Skins)** aus denselben Quellen über Manifest-Dateien.
- **Automatische Sprite-Atlas-Generierung** inklusive Retina-Varianten (`@2x`) und `image-set()`.
- **Cursor-Verwaltung** mit Hotspot, Fallback und Retina-Unterstützung.
- **Vorladen von CSS-Bildern** über eine generierte Preload-Regel.
- **Automatische Bilderzeugung** aus PSD-Entwürfen (Buttons, Icons, App-Icons, Animations-Strips) via μPS.
- **Inkrementeller Build mit Cache** — nur Geändertes wird neu generiert.
- **Aussagekräftige Fehlermeldungen** mit Datei, Zeile und Quelltext-Ausschnitt.

Im Gegensatz zu μCSS 1 läuft Version 2 vollständig in Node.js (ab Version 18) und benötigt weder Photoshop noch andere Adobe-Produkte. Die Build-Steuerung erfolgt über Gulp oder direkt über die Node-API.

1.1 Einordnung: Warum μCSS?

Für fast jeden Teilaspekt von μCSS existiert ein etabliertes Einzelwerkzeug — aber kein System, das alles bündelt:

Funktion	Nächstes existierendes Pendant	Was dort fehlt
Variablen & Farbfunktionen	Sass/LESS (<code>lighten()</code> , <code>mix()</code>), natives CSS (<code>color-mix()</code> , Custom Properties)	kein eingebettetes JavaScript
JavaScript in CSS-Werten	postcss-functions (registrierte JS-Funktionen als CSS-Funktionen)	nur benannte Funktionen — keine freien Ausdrücke, keine Direktiven mit Regel-/Dokument-Zugriff
Stylesheets mit voller JS-Power	vanilla-extract (Stylesheets-in-TypeScript)	umgekehrter Ansatz — das normale CSS-Format geht verloren
Sprite-Atlas aus CSS-Referenzen	spritesmith-Familie, postcss-sprites	weitgehend eingefroren, kein Retina-image-set-Workflow,

Funktion	Nächstes existierendes Pendant	Was dort fehlt
		kein gemeinsamer Cache
Bilderzeugung aus PSD-Entwürfen	nur Bausteine (ag-psd, Asset-Export aus Figma/Sketch)	kein Serien-Rendering mit Ebenenstil-Übertragung, nicht mit dem CSS-Build gekoppelt
Cursor, Preload, Skin-Manifest, Medien-Cache	handgestrickte Build-Skripte	nirgends gebündelt

Ein Teil der ursprünglichen μ CSS-Motivation von 2013 ist heute durch natives CSS gelöst (Custom Properties, `color-mix()`, Nesting) — deshalb verzichtet μ CSS 2 bewusst auf LESS-Unterstützung und Vendor-Prefixes. Der verbleibende Kern ist konkurrenzlos: beliebige JavaScript-Ausdrücke im CSS plus Direktiven mit AST-Zugriff, gekoppelt mit Sprite-Atlas inklusive Retina, PSD-Render-Pipeline und inkrementellem Cache, gesteuert über ein Manifest pro Skin. Und da μ CSS intern eine PostCSS-Pipeline ist, bleibt das gesamte PostCSS-Ökosystem (cssnano, Stylelint, ...) andockbar, statt in Konkurrenz dazu zu stehen.

2 Installation und Start

μCSS und μPS sind als npm-Module `gulp-mu-css` und `gulp-mu-ps` verfügbar. Die Installation erfolgt im eigenen Projekt:

```
npm install gulp-mu-css gulp-mu-ps
```

μCSS hängt von μPS ab (Atlas- und Bilderzeugung) — nicht umgekehrt. Beide Module sind separat verwendbar.

Der typische Einstieg ist ein Gulp-Task, der ein Skin-Manifest baut:

```
// gulpfile.mjs
import gulp from "gulp";
import { BuildSkin } from "gulp-mu-css";

export async function SkinStd() {
  await BuildSkin("skins/src/std.μcss.mjs");
}
export function SkinWatch() {
  gulp.watch(["skins/src/**", "dev/media/final/**"], SkinStd);
}
```

Der Aufruf `npx gulp SkinStd` kompiliert den Skin „std“ in das Verzeichnis `skins/std/`. `BuildSkin` ist idempotent und cache-gestützt: Ein zweiter Aufruf ohne Änderungen ist nach wenigen Millisekunden fertig (siehe Kapitel „Build-Ablauf“).

3 Anwendungsfälle

Die folgenden Abschnitte zeigen die typischen Einsatzszenarien — analog zu den Use Cases des alten μ CSS-Handbuchs, aber in der neuen Syntax.

3.1 Benannte Eigenschaften (Variablen)

Der einfachste Anwendungsfall: Ein Wert wird einmal im Skin-Manifest benannt und an beliebig vielen Stellen verwendet. Im Manifest:

```
// skins/src/std.μcss.mjs
import { DefineSkin } from "gulp-mu-css";

export default DefineSkin({
  vars: {
    textColor: "#ff0000",
    backColor: "#0000ff"
  },
  files: [{ source: "src.μ.css", target: "std.css" }]
});
```

Im Quell-Stylesheet `src.μ.css`:

```
div.mydiv {
  padding: 5px;
  font-size: 24px;
  color: μ($.textColor);
  background-color: μ($.backColor);
  border: 10px μ($.backColor) solid;
  border-radius: 10px;
}
```

Kompiliert nach `std.css`:

```
div.mydiv {
  padding: 5px;
  font-size: 24px;
  color: #ff0000;
  background-color: #0000ff;
  border: 10px #0000ff solid;
  border-radius: 10px;
}
```

Anders als beim alten μ CSS sind keine Platzhalter-Properties und keine `Set*`-Direktiven mehr nötig: Der Wert steht direkt dort, wo er hingehört.

3.2 Berechnungen und Ausdrücke

Der Inhalt von `μ(...)` ist ein beliebiger JavaScript-Ausdruck. Damit sind Berechnungen, Bedingungen und Farboperationen direkt im Stylesheet möglich:

```
td.subheadline {
  border-bottom: 1px dashed μ(Lighten($.selectBaseBrgdColor, 0.7));
}
div.panel {
  padding: μ(Math.floor($.basePadding * 0.2))px;
  background-color: μ(Alpha($.baseBrgdColor, 0.85));
  z-index: μ($.PanelZIndex + 10);
}
div.hint {
  color: μ($.darkMode ? "#e0e0e0" : "#202020");
}
```

Für Operationen, die mehrere Properties erzeugen oder die Regel als Ganzes verändern, gibt es Direktiven (-μ:), zum Beispiel mit eigenen Makro-Funktionen aus dem Manifest:

```
div.menu {
  -μ: Borders($.menuBaseBrngdColor, 1, 0.3, -0.3, -0.3, 0.3);
}
```

Die Direktive ruft die Helper-Funktion `Borders` auf, die vier `border-*`-Properties berechnet und in die Regel einfügt (siehe Kapitel „μ-Auswertungskontext“).

3.3 Mehrere Layouts über Skin-Manifeste

Das alte Template-Compiling (eine Vorlagen-CSS plus je eine `μ.layout.css` pro Layout) wird durch mehrere Manifeste ersetzt, die sich dieselben `μ.css`-Quellen teilen:

```
// skins/src/layout_a.mjs
import { DefineSkin } from "gulp-mu-css";
export default DefineSkin({
  vars: { textColor: "#ff0000", backColor: "#0000ff" },
  files: [{ source: "template.μ.css", target: "layout_a.css" }]
});

// skins/src/layout_b.mjs
import { DefineSkin } from "gulp-mu-css";
export default DefineSkin({
  vars: { textColor: "#ff00ff", backColor: "#00ff00" },
  files: [{ source: "template.μ.css", target: "layout_b.css" }]
});
```

Jedes Manifest erzeugt ein eigenes Ausgabeverzeichnis (`skins/layout_a/`, `skins/layout_b/`) mit eigener CSS, eigenen Bildern und eigenem Build-Cache. Gemeinsame Variablen lassen sich als normales JavaScript-Modul importieren und in beiden Manifesten verwenden.

3.4 Automatische Sprite-Generierung

Eines der Highlights von μCSS (in Version 1 wie in Version 2) ist die automatische Erzeugung von Sprite-Atlanten. Die Direktive `Sprite(url)` registriert das Bild einer Regel für den Atlas:

```
div.loginbutton {
  display: inline-block;
  -μ: Sprite("imgs/aqua/but_login_normal.png");
}
div.loginbutton:hover {
  display: inline-block;
  -μ: Sprite("imgs/aqua/but_login_hover.png");
}
div.helpbutton {
  display: inline-block;
  -μ: Sprite("imgs/aqua/but_help_normal.png");
}
```

Beim Build werden alle registrierten Bilder in ein einziges Atlas-Bild gepackt (`imgs/sprites.png`, dazu `imgs/sprites@2x.png` aus den `@2x`-Quellbildern) und die Regeln umgeschrieben:

```
div.loginbutton {
  display: inline-block;
  background-image: url(imgs/sprites.png);
  background-image: image-set(url(imgs/sprites.png)1x, url(imgs/sprites@2x.png)2x);
  background-repeat: no-repeat;
  background-position: 0px 0px;
  width: 55px;
  height: 55px;
}
```


`width`, `height`, `background-position` und `background-repeat` werden automatisch gesetzt; vorhandene Deklarationen dieser Properties in der Regel werden ersetzt. Anders als beim alten μ CSS entfallen die Vendor-Prefix-Varianten (`-webkit-image-set` usw.) — alle relevanten Browser unterstützen `image-set()` seit Jahren unprefixed.

Identische Quellbilder teilen sich automatisch eine Atlas-Position (Duplikat-Reduktion). Der Atlas wird nur neu gepackt, wenn sich die Bildmenge oder ein Quellbild geändert hat.

3.5 Bilder vorladen (Preload)

Wichtige Bilder (z. B. Hover-Zustände und Cursor) können beim Laden der Seite vorab geladen werden. μ CSS sammelt die Bild-URLs und erzeugt eine Preload-Regel, wenn die Manifest-Option `sprites.preloadRule` gesetzt ist:

```
div.csspreload {
  background-image: url(imgs/sprites.png), url(imgs/general/gui/cursors/zoom.png);
  display: none;
}
```

In der HTML-Seite genügt ein leeres Element mit dieser Klasse:

```
<div class="csspreload"></div>
```

Cursor-Bilder aus den `cursors`-Definitionen werden automatisch in die Preload-Liste aufgenommen.

3.6 Cursor mit Hotspot und Fallback

Cursor werden im Manifest definiert und im Stylesheet per Name verwendet. Definition:

```
cursors: [
  { name: "zoom", fallback: "zoom-in", image: "imgs/general/gui/cursors/zoom.png",
    hotspot: [10, 8] },
  { name: "wait", fallback: "wait", image: "imgs/general/gui/cursors/wait.png", hotspot:
    [12, 12] }
]
```

Verwendung als Direktive oder als Wertform:

```
*.cursor_zoom {
  -μ: Cursor("zoom");
}
a.preview {
  cursor: μ(Cursor("zoom"));
}
```

Die Direktive erzeugt die `url()`-Form mit Hotspot und Fallback sowie — wenn eine `@2x`-Bilddatei existiert — zusätzlich die `image-set()`-Variante:

```
*.cursor_zoom {
  cursor: url(imgs/general/gui/cursors/zoom.png) 10 8, zoom-in;
  cursor: image-set(url(imgs/general/gui/cursors/zoom.png)1x,
    url(imgs/general/gui/cursors/zoom@2x.png)2x) 10 8, zoom-in;
}
```

3.7 Automatische Bilderzeugung (media-Steps)

Was früher die μ CSS-Plugins (ButtonCreator, ApplIconMaker, FileCopy) erledigt haben, übernehmen jetzt die `media`-Einträge im Manifest. Sie laufen vor der CSS-Kompilierung und erzeugen bzw. kopieren alle Medien in das Skin-Verzeichnis:

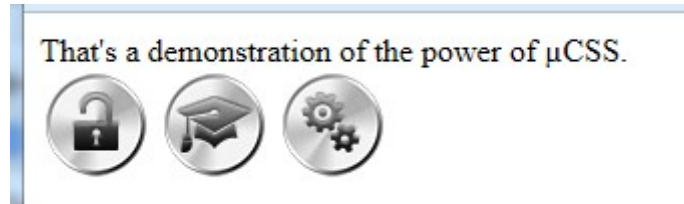
```
media: [
  { buttonsAndIcons: "dev/media/final/general/gui/panelbuttons.psd", layout: "std",
    outputDir: "imgs/general/gui/panelbuttons" },
]
```

```

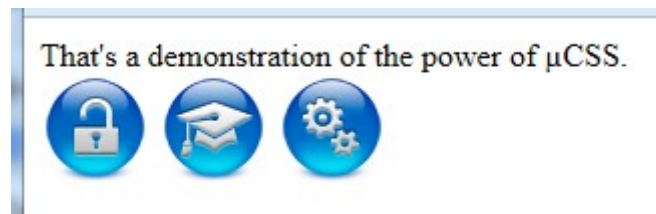
{ buttonsAndIcons: "dev/media/final/general/gui/cursors.psd", layout: "std", outputDir:
"imgs/general/gui/cursors" },
{ appIcons: "dev/media/final/favicon.psd", layout: "std", profiles: ["web"] },
{ sequenceStrip: "dev/media/raw/glittery/imgs", outputFile:
"imgs/general/gui/glittery/glittery.png" },
{ copy: "dev/media/final/general/gui/teaserbgrd.png", to: "imgs/general/gui" },
{ copyFolder: "dev/media/final/fonts", to: "fonts" }
]

```

- **buttonsAndIcons** rendert aus einem geschichteten PSD-Entwurf (Layouts × Icon-Glyphen) komplette Button- und Icon-Serien inklusive @2x-Varianten — der Nachfolger des ButtonCreator-Plugins. Die Fülloptionen (Schlagschatten, Verläufe, Schein usw.) werden von microPS nachgebildet.



Dieselben Icon-Glyphen, gerendert mit zwei Layouts desselben Entwurfsdokuments („alu“ und „aqua“)



Mit mode: "topLayerSets" arbeitet der Step im zweiten Legacy-Modus (CreateByTopLayerSets): Statt der Icon×State-Matrix wird **jede direkte Untergruppe der Layout-Gruppe zu genau einem Bild**, benannt nach dem Gruppennamen. Eine icons-Gruppe ist hier nicht nötig — typisch für Logos, Animationsframes (z. B. activityindicator) und Emoticons. Optional grenzt setPattern (Regex-String) die exportierten Gruppen ein.

```

{ buttonsAndIcons: "dev/media/final/general/activityindicator.psd", layout: "std",
mode: "topLayerSets", outputDir: "imgs/general" }

```

- **appIcons** erzeugt App-Icons und Favicons profilbasiert (web, ios, play) aus einem quadratischen Master — der modernisierte ApplIconMaker.
- **sequenceStrip** baut aus einer Bildsequenz (Einzeldateien oder ein Bild im DSD-Format) einen horizontalen Animations-Strip mit JSON-Frame-Daten:



Animations-Strip, generiert aus 19 Einzel-Frames einer Rauch-Animation

- **copy / copyFolder** kopieren statische Dateien (Make-artig: nur wenn die Quelle neuer ist).

Die Details der Bildgeneratoren (Ebenenstruktur der Entwürfe, DSD-Format, Profile) beschreibt die μPS-Dokumentation.

3.7.1 Zwischenschritte raw → final (`outputBase`)

Standardmäßig schreiben alle Steps in das Skin-Ausgabeverzeichnis. Mit outputBase: "project" schreibt ein Step stattdessen relativ zum Projektstamm — damit lässt sich die Erzeugung von Zwischenergebnissen (z. B. Sequenz-Strips aus dev/media/raw/... nach dev/media/final/...) direkt im Manifest abbilden. Die Steps laufen in Manifest-Reihenfolge, ein nachfolgender copy/copyFolder-Step übernimmt das Ergebnis dann wie jedes andere Final-Asset in den Skin:

```
media: [  
  // 1. Strip aus den Roh-Frames in den final-Baum generieren ...  
  { sequenceStrip: "dev/media/raw/flyex/frames", outputFile:  
    "dev/media/final/general/gui/flyex/flyex.png", outputBase: "project" },  
  // 2. ... und von dort wie gewohnt in den Skin kopieren.  
  { copyFolder: "dev/media/final/general/gui/flyex", to: "imgs/general/gui/flyex",  
    filter: "\\.(png|json)$" }  
]
```

Auch für diese Steps gilt der inkrementelle Build: Generiert wird nur, wenn das Ergebnis fehlt oder sich Konfiguration bzw. Quellen (mtime/Größe) geändert haben. Alternativ kann der raw-→-final-Schritt natürlich weiterhin als eigener Gulp-Task vor dem µCSS-Build laufen — `outputBase: "project"` ist der Weg, wenn alles in einer Manifest-Datei stehen soll.

4 Grundideen von µCSS

4.1 Das .µ.css-Format

Quell-Stylesheets heißen *.µ.css. Das Doppelsuffix sorgt dafür, dass Editoren die Dateien als normales CSS erkennen und das Syntax-Highlighting ohne weitere Konfiguration funktioniert. Für den Compiler ist die Endung irrelevant — das Manifest referenziert die Quellen explizit; wer das µ-Zeichen in Dateinamen vermeiden möchte, kann ebenso *.mu.css verwenden. Inhaltlich ist eine .µ.css-Datei Standard-CSS mit genau zwei Erweiterungspunkten — beide sind syntaktisch valides CSS, so dass Editoren und Linter normal funktionieren:

1. Werte-Interpolation µ(Ausdruck) — überall, wo ein CSS-Wert steht.

2. Direktiven -µ: Ausdruck; — als Deklaration innerhalb einer Regel.

Wer das µ-Zeichen nicht auf der Tastatur hat, verwendet die ASCII-Aliasse µu(...) und -mu: — beide Formen sind gleichwertig.

Anders als beim alten µCSS gibt es **keine Steuer-Regeln** (::µcss-init usw.) mehr im Stylesheet: Alles, was die Kompilierung steuert (Variablen, Cursor-Definitionen, Medienerzeugung, Dateizuordnung), steht im Skin-Manifest — einer normalen JavaScript-Datei (siehe unten).

4.2 Werte-Interpolation µ(Ausdruck)

µ(...) ist aus CSS-Sicht eine unbekannte, aber valide Funktion. Beim Kompilieren wird der Inhalt als JavaScript-Ausdruck ausgewertet und das µ(...)-Vorkommen durch das Ergebnis ersetzt:

```
::selection {
  background-color: µ($.selectBaseBrkdOnDarkColor);
  color: µ($.selectBaseTextColor);
}
```

Mehrere Interpolationen pro Wert sind erlaubt (padding: µ(\$.padY)px µ(\$.padX)px;), ebenso Interpolationen in At-Rule-Parametern (z. B. @media (max-width: µ(\$.breakpoint)px)). Liefert ein Ausdruck null oder undefined, bricht die Kompilierung mit einer Fehlermeldung ab — so fallen Tippfehler in Variablenamen sofort auf.

Diese eine Erweiterung ersetzt die komplette Set*-Familie des alten µCSS (SetColor, SetBackgroundColor, SetZIndex, SetWidth, AddProperty für einfache Werte usw.): Die Property steht wieder an ihrem Platz, ein Platzhalter-Wert ist nicht mehr nötig.

4.3 Direktiven -µ: Ausdruck

Direktiven sind Deklarationen mit dem Property-Namen -µ (oder -mu). Ihr Wert ist ein einzelner JavaScript-Ausdruck, der mit Zugriff auf die umgebende Regel und das Dokument ausgeführt wird. Die Direktive selbst wird aus der Ausgabe entfernt:

```
div.panel.modal div.companylogo {
  -µ: Sprite("imgs/logos/dosing_logo.png");
  margin-left: auto;
}
*.cursor_zoom {
  -µ: Cursor("zoom");
}
div.content.glittery {
  -µ: Sprite("imgs/general/gui/glittery/glittery.png", { afterWork: GlitterySprite });
}
```

Das µ.-Präfix des alten µCSS entfällt — der Kontext ist implizit. Direktiven werden in Dokumentreihenfolge ausgewertet.

4.4 Das Skin-Manifest

Pro Skin (CSS-Thema) liegt im Quellverzeichnis eine Manifest-Datei `<skinname>.μcss.mjs` — normales JavaScript (ES6+), importierbar und testbar. Das Doppelsuffix `.μcss.mjs` (ASCII-Alternative `.mucss.mjs`) kennzeichnet die Datei eindeutig als `μCSS`-Skin-Manifest, damit sie nicht mit einem gewöhnlichen Modul oder Gulpfile verwechselt wird; der Skin-Name ergibt sich aus dem Teil davor (`std.μcss.mjs` → Skin `std`). Sie ersetzt die alte Steuerdatei `μ.std.css` vollständig:

```
// skins/src/std.μcss.mjs – Skin "std", Ziel: skins/std/
import { DefineSkin } from "gulp-mu-css";
import { GlitterySprite, FlyEx, Borders, TableBackgrounds } from "./helpers.mjs";

export default DefineSkin({
  vars: {
    baseBrgdColor: "#202020",
    selectBaseBrgdColor: "#007570",
    PanelZIndex: 9000
  },
  helpers: { GlitterySprite, FlyEx, Borders, TableBackgrounds },
  cursors: [
    { name: "zoom", fallback: "zoom-in", image: "imgs/general/gui/cursors/zoom.png",
    hotspot: [10, 8] }
  ],
  media: [
    { buttonsAndIcons: "dev/media/final/general/gui/panelbuttons.psd", layout: "std",
    outputDir: "imgs/general/gui/panelbuttons" },
    { copyFolder: "dev/media/final/fonts", to: "fonts" }
  ],
  imageFormat: "png",
  sprites: { file: "imgs/sprites.png", retina: true, preloadRule: true },
  files: [
    { source: "src.μ.css", target: "std.css" },
    { source: "src_tinymce.μ.css", target: "std_tinymce.css" }
  ]
});
```

Der Skin-Name ergibt sich aus dem Dateinamen des Manifests, das Ausgabeverzeichnis aus dem Skin-Namen. Die vollständige Feld-Referenz steht im Kapitel „Manifest-Referenz“.

4.5 JavaScript ohne Ersatzzeichen

Bei `μCSS` 1 mussten die Zeichen `{`, `}` und `;` in JavaScript-Anweisungen durch `«`, `»` und `;` ersetzt werden, um CSS-Syntaxkonflikte zu vermeiden. Dieses Muster entfällt in Version 2 ersatzlos:

- In `μ(...)` und `-μ: ...` stehen **einzelne Ausdrücke** — der Parser zählt Klammern und berücksichtigt Strings, so dass auch Objekt-Literale (`{ afterWork: ... }`) und verschachtelte Aufrufe problemlos funktionieren.
- **Mehrzeilige Logik** (Schleifen, Funktionsdefinitionen) gehört nicht ins Stylesheet, sondern in die Helper-Module des Manifests — echte `.mjs`-Dateien mit Syntax-Highlighting, Linting und Debugger.

5 Build-Ablauf und Gulp-Integration

Das alte `µCSS` wurde über ein modales Dialogfenster in Photoshop bedient. An dessen Stelle treten `BuildSkin` und `Gulp`: Der Build ist ein normaler, skriptbarer Node-Prozess — geeignet für Watch-Modus, CI und Automatisierung.

5.1 Verzeichniskonventionen

Für ein Manifest `skins/src/std.µcss.mjs` gilt (alle Pfade überschreibbar):

Pfad	Bedeutung
<code>skins/src/</code>	Quellverzeichnis: Manifeste, <code>.µ.css</code> -Dateien, Helper-Module
<code>skins/std/</code>	Ausgabeverzeichnis des Skins „std“: CSS, Bilder, Fonts, Sounds
<code>skins/std/.cache/build.json</code>	Build-Cache des Skins (Fingerprints, Atlas-Positionen)
Projektstamm	Zwei Ebenen über dem Manifest; Basis für media-Quellpfade wie <code>dev/media/...</code>

`BuildSkin(manifestPfad, optionen)` akzeptiert `{ outputDir, rootDir, force }` zum Übersteuern.

5.2 Kompilierungs-Ablauf

Ein `BuildSkin`-Lauf führt fünf Schritte aus:

- 1. media-Steps:** Alle Bilder werden erzeugt bzw. kopiert (`microPS`) — sie müssen existieren, bevor Atlas und Cursor-Prüfungen laufen.
- 2. Kompilierung:** Jede `files`-Quelle wird geparkt (`PostCSS`); Direktiven und Interpolationen werden in Dokumentreihenfolge ausgewertet. `Sprite()`-/`Cursor()`-Aufrufe registrieren zunächst nur ihre Bildreferenzen.
- 3. Sprite-Atlas:** Alle registrierten Bilder werden gepackt (inkl. `@2x`); danach werden die betroffenen Regeln umgeschrieben.
- 4. Hooks:** `afterWork`-Callbacks laufen mit Atlas-Ergebnis und AST-Zugriff.
- 5. Ausgabe:** Die kompilierten CSS-Dateien werden in das Skin-Verzeichnis geschrieben, der Cache wird gespeichert.

5.3 Inkrementeller Build und Cache

Jeder Lauf erzeugt nur das neu, was sich tatsächlich geändert hat. Primärer Mechanismus sind Datei-Modifikations-Zeitstempel (`mtime` plus Dateigröße) — bei großen PSD-Quellen um Größenordnungen schneller als Inhalts-Hashes:

- **Generator-Steps** (`buttonsAndIcons`, `appIcons`, `sequenceStrip`) werden übersprungen, wenn ihre Konfiguration unverändert ist, alle Quelldateien unveränderte Fingerprints haben und alle Ausgabedateien noch existieren. PSD-Rendering ist der teuerste Posten des Builds — hier zahlt der Cache am meisten.
- `copy/copyFolder` arbeiten Make-artig: kopiert wird nur, wenn die Quelle neuer als das Ziel ist.

- **Der Sprite-Atlas** wird nur neu gepackt, wenn sich die Bildmenge oder ein Quellbild (inkl. @2x) geändert hat — sonst werden die gespeicherten Positionen wiederverwendet und nur die CSS-Regeln damit umgeschrieben.
- **Die CSS-Kompilierung** selbst ist billig und läuft im Zweifel immer.

Der Cache liegt pro Skin in `<outputDir>/cache/build.json` und trägt die Versionsnummern von `μCSS` und das Cache-Schema; bei Abweichung wird er verworfen (Vollbuild). `BuildSkin(manifest, { force: true })` oder das Löschen von `cache/` erzwingt den Vollbuild explizit.

5.4 Gulp-Tasks und Watch

Ein Task pro Skin genügt; `BuildSkin` ist re-entrant:

```
import gulp from "gulp";
import { BuildSkin } from "gulp-mu-css";

export async function SkinStd() {
  const report = await BuildSkin("skins/src/std.μcss.mjs");
  console.log(`${report.skin}: ${report.files.length} Dateien, Atlas $
{report.atlasSkipped ? "aus Cache" : "neu gepackt"}, ${report.duration} ms`);
}
export function SkinWatch() {
  gulp.watch(["skins/src/**/*.μ.css", "skins/src/*.mjs", "dev/media/final/**"], SkinStd);
}
```

Der Rückgabewert von `BuildSkin` ist ein Report mit `skin`, `outputDir`, `media` (Step-Ergebnisse mit `skipped-Flag`), `files`, `atlas`, `atlasSkipped` und `duration`.

6 Manifest-Referenz (DefineSkin)

`DefineSkin(konfiguration)` deklariert die Skin-Konfiguration (Default-Export des Manifests) und validiert die Grundstruktur. Alle Felder sind optional, sofern nicht anders angegeben.

6.1 vars

Objekt mit den Skin-Variablen — der Ersatz für `μ.$.*`. In `μ.css`-Ausdrücken als `$.name` zugreifbar, in Helper-Funktionen als `this.$.name`. Werte sind beliebige JavaScript-Werte (Strings, Zahlen, Objekte, auch berechnete).

```
vars: { baseBrgdColor: "#202020", PanelZIndex: 9000, icon_pencil: "\\e91c" }
```

6.2 helpers

Objekt mit benutzerdefinierten Funktionen (Makros und Hooks). Sie stehen in `μ.css`-Ausdrücken unter ihrem Namen zur Verfügung. Funktionen, die mit `function` (nicht als Arrow-Funktion) deklariert sind, erhalten beim Aufruf `this` = Auswertungs-Scope (siehe Kapitel „μ-Auswertungskontext“).

```
import { Borders, GlitterySprite } from "../helpers.mjs";  
// ...  
helpers: { Borders, GlitterySprite }
```

6.3 cursors

Liste der Cursor-Definitionen — der Ersatz für `μ.DefCursor`:

Feld	Default	Bedeutung
<code>name</code>	— (Pflicht)	Name, unter dem der Cursor mit <code>Cursor("name")</code> verwendet wird.
<code>fallback</code>	<code>= name</code>	Standard-Cursor-Name (W3C) als Fallback.
<code>image</code>	<code>""</code>	Bild-URL relativ zum Skin-Verzeichnis; leer = Definition nur als Fallback-Mapping.
<code>hotspot</code>	<code>[0, 0]</code>	Klickpunkt <code>[x, y]</code> im Bild; <code>0,0</code> wird in der Ausgabe weggelassen.
<code>forceFallback</code>	<code>false</code>	Hängt den Fallback zusätzlich als letzte cursor-Deklaration an.

Cursor-Bilder werden automatisch in die Preload-Liste aufgenommen.

6.4 media

Liste der Medien-Steps; sie laufen in der angegebenen Reihenfolge vor der Kompilierung. Der Step-Typ ergibt sich aus dem Schlüsselfeld:

Step	Pflichtfelder	Weitere Felder	Wirkung
<code>buttonsAndIcons</code>	<code>Quell-PSD</code> , <code>layout</code> , <code>outputDir</code>	<code>retina</code> (Default <code>true</code>), <code>format</code> , <code>mode</code> , <code>setPattern</code>	Button-/Icon-Serien aus einem Entwurfsdokument

Step	Pflichtfelder	Weitere Felder	Wirkung
			rendern (microPS ButtonAndIconCreator) .mode: "topLayerSets" schaltet auf „ein Bild pro Top-Untergruppe“ um (Legacy-CreateByTopLayerSets, ohne icons-Gruppe — für Logos, Animationsframes, Emoticons); setPattern (Regex-String) filtert dabei die exportierten Gruppen.
appIcons	Quell-PSD/PNG	outputDir, profiles, layout, background, appName, shortName, themeColor	App-Icons/Favicons profilbasiert generieren (microPS ApplconMaker).
sequenceStrip	Quelle (Ordner oder DSD-Bild), outputFile	retina (Default true), writeMapFile (Default true), format	Horizontalen Animations-Strip plus JSON-Frame-Daten erzeugen (microPS SequenceStrip).
copy	Quelldatei	to (Zielordner, Default Skin-Wurzel)	Einzeldatei kopieren, wenn die Quelle neuer ist.
copyFolder	Quellordner	to (Default = Ordnername), filter (Regex-String, z. B. "\\.(woff2? ttf)\$")	Ordner rekursiv kopieren (Make-artig), optional gefiltert.

Quellpfade sind relativ zum Projektstamm, Zielpfade relativ zum Skin-Verzeichnis. Jeder Step akzeptiert zusätzlich outputBase: "skin" (Default) oder "project": Mit "project" ist der Zielpfad relativ zum Projektstamm — für Zwischenschritte wie raw → final (siehe Kapitel „Automatische Bilderzeugung“).

6.5 imageFormat

Globaler Bildformat-Schalter: "png" (Default) oder "webp". **Wirkungsbereich:**

- **Bilderzeugende media-Steps** (buttonsAndIcons, appIcons*, sequenceStrip) und der **Sprite-Atlas** (sofern sprites.format nicht gesetzt ist, siehe unten).
- Einzelne Steps können das Format per eigenem format-Feld übersteuern.
- Direkt referenzierte Bestandsbilder (url(...)) in den Quellen, copy-/copyFolder-Steps) bleiben unangetastet — ein copyFolder-filter kopiert nur, was er durchlässt. Schließt der filter die aktive imageFormat-Endung aus (z. B. "\\.(png|json)\$" bei imageFormat: "webp"), gibt der Build eine **Warnung** aus, statt die gerade erzeugten Dateien stillschweigend zu verwerfen.

`**appIcons` erzeugt plattformspezifische Icon-Sätze (PNG, teils ICO) und **ignoriert** `imageFormat` **bewusst** — App-/Favicon-Formate sind festgelegt.*

6.6 sprites

Optionen des Sprite-Atlas — der Ersatz für `μ.options.sprites.*`:

Feld	Default	Bedeutung
<code>file</code>	<code>"imgs/sprites.png"</code>	Atlas-Datei (URL, wie sie in der CSS erscheint).
<code>format</code>	(entfällt)	Atlas-Ausgabeformat (" <code>png</code> "/" <code>webp</code> "), unabhängig von <code>imageFormat</code> . Gesetzt, überschreibt es die aus <code>imageFormat</code> abgeleitete Endung; sonst gilt <code>imageFormat</code> . So lässt sich allein der Atlas auf WebP umstellen (<code>sprites: { file: "imgs/sprites.png", format: "webp" }</code>), ohne die Generator-Steps anzufassen — die <code>Sprite()</code> -Quellbilder dürfen weiterhin PNG sein.
<code>retina</code>	<code>true</code>	Erzeugt zusätzlich <code><name>@2x</code> aus den <code>@2x</code> -Quellbildern (müssen neben den 1x-Quellen liegen).
<code>padding</code>	<code>0</code>	Abstand in Pixeln zwischen den Sprites.
<code>preloadRule</code>	<code>false</code>	Erzeugt die <code>div.csspreload</code> -Regel im ersten Stylesheet.

Auflösung der `Sprite()`-Quellbilder: Der in `Sprite("...")` referenzierte Pfad wird formatunabhängig aufgelöst — zuerst der literale Pfad, dann derselbe Name mit einer der unterstützten Raster-Endungen (`png`, `webp`). Liegt ein generiertes Quellbild also als PNG vor, obwohl die CSS-Referenz `.webp` nennt (oder umgekehrt), bricht der Build nicht ab. Ein Fehler entsteht nur, wenn **keine** Variante existiert. Die `@2x`-Variante muss dieselbe Endung wie das aufgelöste 1x-Bild haben.

6.7 files

Liste der zu kompilierenden Stylesheets — der Ersatz für `μ.DependentCSSFile`. `source` ist relativ zum Quellverzeichnis, `target` relativ zum Skin-Verzeichnis; beide Felder sind Pflicht:

```
files: [  
  { source: "src.μ.css", target: "std.css" },  
  { source: "src_tinymce.μ.css", target: "std_tinymce.css" }  
]
```

Alle Dateien eines Skins teilen sich Variablen, Helpers und den Sprite-Atlas. Die Preload-Regel landet im ersten Stylesheet.

7 μ -Auswertungskontext (Referenz)

Jeder $\mu(\dots)$ -Ausdruck und jede $-\mu$ -Direktive wird in einem Scope ausgewertet, der die folgenden Bindungen enthält. Er entspricht dem alten globalen μ -Objekt — nur ohne Präfix.

7.1 Das \$-Objekt

\$ enthält die vars des Manifests. Lesen und Schreiben ist möglich; Änderungen gelten für den weiteren Verlauf der Kompilierung (Dokumentreihenfolge, über alle files eines Skins hinweg).

```
div.box { z-index:  $\mu$ ($.PanelZIndex + 1); }
```

7.2 Farb- und Wert-Funktionen

Funktion	Beschreibung
Lighten(color, step, model = "hsl")	Hellet eine Farbe relativ auf (positiver step) oder dunkelt sie ab (negativer step): $L' = \text{clamp}(L + L \cdot \text{step})$. Das Modell "hsl" ist bitgenau zum alten μ CSS; "oklch" skaliert wahrnehmungsgleichmäßig. Alpha bleibt erhalten.
Alpha(color, alpha)	Ersetzt den Alpha-Kanal einer Farbe. alpha nach den Regeln aus „Arbeiten mit Farben“.
MixColors(color1, color2)	Kanalweiser Mittelwert zweier Farben (inklusive Alpha).
AlphaValue(alpha)	Wandelt eine Alpha-Angabe in ein Byte (0–255) um.
ParseColor(color)	Parst eine CSS-Farbe in die interne 32-Bit-Darstellung 0xAARRGGBB.
FormatColor(color, alphaDecimals = 3)	Serialisiert die interne Darstellung zurück zu CSS (#rrggbb bzw. rgba(...)).
PxUnit(value)	Zahlen werden zu "<n>px", alles andere (z. B. calc()-Strings) wird unverändert durchgereicht.

7.3 Regel- und Dokument-Methoden

Innerhalb von Direktiven (und Helper-Funktionen mit this-Bindung) stehen zusätzlich die Manipulations-Methoden der umgebenden Regel bereit:

Methode / Eigenschaft	Beschreibung
AddProperty(name, value, important?)	Hängt eine Deklaration an die Regel an (Duplikate erlaubt — für Fallback-Ketten wie mehrfache background-image).
ChangeProperty(name, value, important?)	Ändert alle Deklarationen der Property bzw. legt sie an, wenn sie fehlt.
RemoveProperty(name)	Entfernt alle Deklarationen der Property.
AddRule(selector)	Fügt eine neue Regel am Dokumentende an; Rückgabe ist die Regel (mit denselben Methoden).

Methode / Eigenschaft	Beschreibung
<code>InsertRule(selector)</code>	Fügt eine neue Regel direkt hinter der aktuellen Regel ein; aufeinanderfolgende Aufrufe behalten ihre Reihenfolge. Ersatz für das alte <code>AddBlock(n, μ.elementNo)</code> .
<code>rule</code>	Die umgebende Regel als <code>CssRule</code> -Objekt (<code>rule.selector</code> , <code>rule.GetProperty(...)</code> , ...).
<code>document</code>	Das gesamte Stylesheet als <code>CssDocument</code> — z. B. für Pfad-Adressierung: <code>document.FindRule("@keyframes glittery", "from")</code> . Ersatz für die alten <code>RememberBlocks</code> .

7.4 Sprite()

Registriert das Bild der Regel für den Sprite-Atlas (nur als Direktive):

```
-μ: Sprite(url, optionen?);
```

Parameter / Option	Default	Beschreibung
<code>url</code>	— (Pflicht)	Bild-URL relativ zum Skin-Verzeichnis.
<code>offsetWidth</code>	0	Wird zur berechneten <code>width</code> addiert.
<code>offsetHeight</code>	0	Wird zur berechneten <code>height</code> addiert.
<code>offsetPosX</code>	0	Wird zur <code>background-position-X</code> -Koordinate addiert.
<code>offsetPosY</code>	0	Wird zur <code>background-position-Y</code> -Koordinate addiert.
<code>afterWork</code>	—	Hook-Funktion, läuft nach der Atlas-Auflösung (siehe unten).

Beim Atlas-Auflösen werden `background-image (url(...) plus image-set(...) bei Retina)`, `background-repeat`, `background-position`, `width` und `height` der Regel gesetzt; vorhandene Deklarationen dieser Properties werden ersetzt.

7.5 Cursor()

Wendet eine Cursor-Definition aus dem Manifest an — als Direktive (`-μ: Cursor("zoom");`, schreibt die `cursor`-Deklarationen der Regel um) oder als Wertform (`cursor: μ(Cursor("zoom"))`); liefert nur den `url(...) x y, fallback-String`). Unbekannte Namen werden unverändert durchgereicht — Standard-Cursor wie `pointer` funktionieren so ohne Definition.

7.6 Helper-Funktionen und this-Bindung

Helper aus dem Manifest, die mit `function` deklariert sind, werden mit `this` = Auswertungs-Scope aufgerufen. Damit lassen sich Makros im Stil der alten `μ.$.`Funktionen schreiben — nur als normales, testbares JavaScript:

```
// helpers.mjs
```

```
import { Lighten } from "gulp-mu-css";

export function Borders(_baseColor, _pixelWidth, _topLighten, _rightLighten,
_bottomLighten, _leftLighten) {
  this.AddProperty("border-top", `${_pixelWidth}px solid ${Lighten(_baseColor,
_topLighten)}`);
  this.AddProperty("border-right", `${_pixelWidth}px solid ${Lighten(_baseColor,
_rightLighten)}`);
  this.AddProperty("border-bottom", `${_pixelWidth}px solid ${Lighten(_baseColor,
_bottomLighten)}`);
  this.AddProperty("border-left", `${_pixelWidth}px solid ${Lighten(_baseColor,
_leftLighten)}`);
}
```

Über `this` sind alle Scope-Bindungen erreichbar: `this.AddProperty(...)`, `this.InsertRule(...)`, `this.rule`, `this.document` und `this.$`. Die Bindung bleibt auch erhalten, wenn ein Helper als `afterWork`-Wert an `Sprite()` übergeben wird. Arrow-Functions haben kein eigenes `this` und eignen sich daher nur für Helpers ohne Regel-Zugriff.

Portierte Referenz-Makros des AiDPix-Projekts (`Borders`, `TableBackgrounds`, `GlitterySprite`, `FlyEx`, `FlyExUtils`) liegen im `μCSS-Repository` unter `test/fixtures/aidpix-helpers.mjs`.

7.7 afterWork-Hooks

Der `afterWork`-Hook einer `Sprite`-Registrierung läuft, nachdem der Atlas gepackt und die Regel umgeschrieben wurde. Er erhält einen Kontext mit allen Ergebnisdaten:

Feld	Beschreibung
<code>rule</code>	Die umgeschriebene Regel (<code>CssRule</code>).
<code>document</code>	Das Stylesheet (<code>CssDocument</code>).
<code>url</code>	Die registrierte Bild-URL.
<code>baseDir</code>	Skin-Verzeichnis (zum Auflösen von Nachbar-Dateien, z. B. <code>.json-Maps</code>).
<code>sprite</code>	<code>{ x, y, width, height }</code> — Position und Größe im Atlas.
<code>atlas</code>	<code>{ file, retinaFile, width, height }</code> — die Atlas-Dateien und -Gesamtgröße.

Typischer Einsatz: Animations-Keyframes aus den Frame-Daten eines `sequenceStrip` generieren (siehe `GlitterySprite` in den Referenz-Makros).

8 Arbeiten mit Farben

8.1 Farben definieren

Die interne Darstellung einer Farbe ist ein vorzeichenloser 32-Bit-Integer der Form `0xAARRGGBB` (Alpha, Rot, Grün, Blau; je 8 Bit). In allen Farb-Funktionen sind folgende Eingabeformen möglich:

- **32-Bit-Integer** wie die interne Darstellung, z. B. `0xffff0000` für deckendes Rot.
- **#-Notation** mit zwei Hex-Ziffern pro Kanal, z. B. `"#00ff00"` (Alpha wird auf deckend gesetzt).
- **Kurze #-Notation** mit einer Hex-Ziffer pro Kanal, z. B. `"#0f0"`.
- **Erweiterte #-Notation** mit Alpha-Kanal als vierte Komponente, z. B. `"#0000ff80"` für halbtransparentes Blau.
- **rgb()-Notation**, absolut oder prozentual, z. B. `"rgb(255,0,0)"` oder `"rgb(100%,0%,0%)"`.
- **rgba()-Notation** mit Float-Alpha, z. B. `"rgba(255,0,0,0.5)"`.
- **CSS-Farbnamen** (W3C CSS Color Module, erweiterte Liste), z. B. `"red"`, `"teal"`, `"rebeccapurple"` sowie `"transparent"`.

Werte außerhalb des gültigen Bereichs werden begrenzt (geclippt).

8.2 Alpha-Werte

Alpha-Angaben (z. B. für `Alpha(color, a)`) sind in mehreren Formen möglich:

- **Float** zwischen `0.0` (transparent) und `1.0` (deckend).
- **Integer** zwischen `0` und `255`.
- **Hex-String**, z. B. `"0x80"`.
- **Prozent-String** von `"0%"` bis `"100%"`.
- **Schlüsselwörter** `"transparent"` (0), `"translucent"` (128) und `"opaque"` (255).

8.3 Farbberechnungen

Die Funktionen `Lighten`, `Alpha` und `MixColors` (Kapitel „μ-Auswertungskontext“) decken die typischen Berechnungen ab. Deckende Ergebnisse werden als `#rrggbb` ausgegeben, transparente als `rgba(r,g,b,a)` mit drei Alpha-Nachkommastellen — identisch zum alten μCSS.

```
div.menu {
  background-color: μ(Alpha($.baseBrgdColor, 0.9));
  border-top: 1px solid μ(Lighten($.baseBrgdColor, 0.3));
  border-bottom: 1px solid μ(Lighten($.baseBrgdColor, -0.3));
}
```

Für neue Skins empfiehlt sich das `"oklch"`-Modell von `Lighten`: Es verändert die wahrgenommene Helligkeit gleichmäßig über alle Farbtöne, während das Legacy-Modell `"hsl"` bei gesättigten Farben sichtbar springen kann.

9 Node-API

Neben dem Manifest-Workflow ist jede Schicht von μ CSS auch direkt als Node-API nutzbar — etwa für eigene Gulp-Transformationen oder Tests.

Export	Beschreibung
<code>CompileMcCss(source, options)</code>	Kompiliert <code>.μ.css</code> -Quelltext zu einem <code>CssDocument</code> . Optionen: <code>vars</code> , <code>helpers</code> , <code>from</code> (Dateiname für Fehlermeldungen), <code>context</code> (gemeinsamer <code>MuContext</code>), <code>sprites</code> (<code>SpriteManager</code>), <code>cursors</code> (<code>CursorManager</code>).
<code>CssDocument</code> / <code>CssRule</code>	JSON-fähiger Wrapper über den PostCSS-AST: <code>FromFile/FromString</code> , <code>FindRule(...pfad)</code> , <code>FindRules(selektorOderRegex)</code> , <code>AddRule</code> , <code>GetProperty/AddProperty/ChangeProperty/RemoveProperty</code> , <code>ToCss()</code> , <code>ToFile()</code> , <code>ToJson()/FromJson()</code> . Für Spezialfälle ist der rohe PostCSS-AST über <code>document.root</code> zugänglich.
<code>SpriteManager</code>	Sprite-Registrierung und Atlas-Auflösung: <code>new SpriteManager({ baseDir, atlasFile, retina, padding, preloadRule, preload, writeMapFile })</code> , <code>Register(rule, url, optionen)</code> , <code>await Resolve(document)</code> .
<code>CursorManager</code>	Cursor-Definitionen: <code>new CursorManager(definitionen, { baseDir, preload })</code> , <code>Apply(rule, name)</code> , <code>Value(name)</code> .
<code>PreloadRegistry</code>	Sammelt Bild-URLs und erzeugt die <code>div.csspreload</code> -Regel.
<code>DefineSkin(config)</code> / <code>BuildSkin(manifest, options)</code>	Manifest-Deklaration und Skin-Build (siehe Kapitel „Build-Ablauf“).
<code>MuContext</code>	Auswertungskontext für eigene Pipelines: <code>new MuContext({ vars, helpers })</code> , <code>Evaluate(Quelltext, extraScope)</code> .
<code>Lighten</code> , <code>Alpha</code> , <code>AlphaValue</code> , <code>MixColors</code> , <code>ParseColor</code> , <code>FormatColor</code> , <code>PxUnit</code>	Die Farb- und Wert-Funktionen als direkte Importe.

Beispiel einer JSON-basierten Gulp-Manipulation:

```
import { CssDocument } from "gulp-mu-css";

const doc = await CssDocument.FromFile("skins/src/src. $\mu$ .css");
doc.FindRules(/^div\.panel/).forEach((_rule) => _rule.AddProperty("outline", "none"));
doc.FindRule("@keyframes glittery", "from").ChangeProperty("background-position-x", "-128px");
const json = doc.ToJson();
await doc.ToFile("out/std.css");
```

10 Fehlerdiagnostik

Build-Fehler nennen immer den Verursacher samt Quellbezug:

- **Inline-JavaScript** ($\mu(\dots)$ -Interpolationen und $-\mu$:-Direktiven): Fehler werden als PostCSS-Fehler mit Datei, Zeile, Spalte und Quelltext-Ausschnitt gemeldet. Bei mehreren $\mu(\dots)$ in einem Wert wird der fehlschlagende Ausdruck genannt. JavaScript-Syntaxfehler erscheinen als `invalid JavaScript expression "<Quelltext>" (...)`.

```
CssSyntaxError: skins/src/std.μ.css:2:2: microCSS: μ(NopeFn(1)): NopeFn is not defined
```

```
1 | div.b {  
> 2 |     border: 1px solid μ(NopeFn(1));  
   |           ^  
3 | }
```

- **Fehlende Sprite-Bilder**: Vor dem Packen des Atlas wird die Existenz aller Quellbilder (inklusive @2x bei `retina: true`) geprüft. Alle fehlenden Dateien werden gesammelt in einem Fehler gemeldet — jeweils mit URL, aufgelöstem Pfad und der referenzierenden Regel:

```
SpriteManager: 2 sprite image(s) not found:  
- "imgs/nope.png" -> C:\...\skins\std\imgs\nope.png - selector "div.bad1" (skin.μ.css:2)  
- "imgs/also nope.png" -> C:\...\skins\std\imgs\also nope.png - selector "div.bad2"  
(skin.μ.css:3)
```

- **afterWork-Hooks**: Fehler im Hook werden mit Sprite-URL und Regel-Quellposition ummantelt; der Original-Fehler bleibt als `cause` erhalten.
- **Fehlende Cursor-Bilder** erzeugen nur eine Warnung (einmal pro Cursor), da die CSS über den Fallback-Cursor funktionsfähig bleibt.
- **media-Steps**: Fehler nennen Step-Nummer und -Typ, z. B. `BuildSkin: media step 3 of 7 (buttonsAndIcons: "dev/media/buttons.psd") failed:` Fehlende Quellen werden vor der Ausführung geprüft: `copy/copyFolder` und Generator-Steps melden den aufgelösten Pfad statt eines rohen Dateisystemfehlers — bei `copyFolder` mit dem Hinweis, dass vermutlich der erzeugende Schritt (z. B. Sequenzbild-Generierung nach `dev/media/final/...`) nicht gelaufen ist.
- **files-Einträge**: Fehlende Quelldateien werden mit Eintrag und aufgelöstem Pfad gemeldet, bevor kompiliert wird.

11 Migration von µCSS

Für Bestandsprojekte existiert ein Konverter-Werkzeug (`tools/convert-mucss.mjs` im µCSS-Repository), das die alte Syntax mechanisch übersetzt:

Alt (µCSS 1)	Neu (µCSS 2)
<code>-µcss: µ.Cursor("wait");</code> plus <code>cursor: wait;</code>	<code>cursor: µ(Cursor("wait"));</code>
<code>-µcss: µ.SetBackgroundColor(µ.\$.x);</code> plus Platzhalter	<code>background-color: µ(\$.x);</code>
<code>-µcss: µ.AddProperty("border", "1px solid " + µ.Lighten(...));</code>	<code>border: 1px solid µ(Lighten(...));</code>
<code>-µcss: µ.Sprite("p.png");</code>	<code>-µ: Sprite("p.png");</code>
<code>-µcss: µ.SetRememberBlock("n");</code>	entfällt — Pfad-Adressierung: <code>document.FindRule("@keyframes x", "from")</code>
<code>-µcss: //µ.X(...) (deaktiviert)</code>	<code>/* -µ: X(...) */</code>
<code>µ.\$.name = wert (in µ.std.css)</code>	vars-Eintrag im Manifest
<code>µ.DefCursor(...)</code>	cursors-Eintrag im Manifest
<code>µ.plugins.* / FileCopy</code>	media-Einträge im Manifest
<code>µ.\$.Fn = function(...)«...»</code>	exportierte Funktion in <code>helpers.mjs</code>

Die «»i-Funktionskörper werden zu normalem JavaScript zurückübersetzt und landen als Startpunkt in `helpers.mjs`; manuelle Nacharbeit ist hier eingeplant.

raw → final automatisch: Das alte µCSS kopierte nur die bereits fertig erzeugten final-Ordner in den Skin (z. B. `flyex`, `glittery`); die Strips selbst entstanden außerhalb (`SpriteTools`). Der Konverter erkennt solche `CopyFolder2Skin`-Aufrufe auf `dev/media/final/<...>/<name>` und stellt — falls eine passende Quelle `dev/media/raw/<name>/imgs` existiert — automatisch den passenden `sequenceStrip`-Step mit `outputBase: "project"` davor: eine **nummerierte PNG-Frame-Sequenz** (z. B. `glittery`) wird zu einem Ordner-Strip, **einzeln benannte Bilder** (z. B. die DSD-Bilder `flyex.png`/`flyexutils.png`) je zu einem DSD-Strip. So baut die neue Pipeline `final` reproduzierbar aus `raw` und der nachfolgende `copyFolder` übernimmt das Ergebnis in den Skin.

Der Konverter wurde am vollständigen AiDPix-Bestand validiert: Ein Vergleichswerkzeug (`tools/compare-aidpix.mjs`) baut den konvertierten Skin und vergleicht ihn regel- und property-weise gegen die alte kompilierte Ausgabe — Ergebnis: 2 951 Regeln, 0 unerwartete Differenzen (53 dokumentierte Drift-Fälle, z. B. nach dem letzten µCSS-Lauf editierte Quellen).

Bewusst **nicht** übernommen wurden aus dem alten µCSS:

- **Vendor-Prefixes** (`-webkit-`/`-moz-`/`-ms-`--Duplikate): Alle genutzten Features sind seit Jahren unprefixed verfügbar. Vendor-spezifische Selektoren ohne Standard-Pendant (z. B. `::-webkit-scrollbar`) werden natürlich unverändert durchgereicht.
- **Photoshop-Dialogfenster** im Build: Interaktive Parameter gehören in das Manifest oder in Umgebungsvariablen.
- **FTP-Synchronisation**: Dafür gibt es heute etablierte Werkzeuge (CI/CD, `rsync`, `gulp`-Plugins).
- **Die Ersatzzeichen** «»i: siehe Kapitel „Grundideen“.

12 Versionshistorie

Datum	Version	Anmerkungen
2013	1.0	Ursprüngliches µCSS als Adobe-Photoshop-Script: <code>-µcss:-</code> Direktiven, Sprite-Atlas, PSD-Plugins, Steuerung über <code>µ.std.css</code> .
2026-06	2.0.0	Vollständige Node.js-Neuimplementierung (npm-Paket <code>gulp-mu-css</code> , ohne Adobe-Abhängigkeit): Core-Pipeline (M1), Sprites & Cursor (M2), Manifest & Build mit inkrementellem Cache (M3), Hooks & Makros (M4), AiDPix-Migration mit Konverter und Abnahmetest (M5), Handbuch (M6).
2026-06	2.1.0	Atlas-Format unabhängig vom globalen <code>imageFormat</code> über <code>sprites.format</code> ; formatunabhängige Auflösung der <code>Sprite()</code> -Quellbilder (<code>png/webp</code>); Warnung, wenn ein <code>copyFolder-filter</code> das aktive Bildformat ausschließt.
2026-06	2.2.0	Normalisierung veralteter Gradient-Richtungen beim Kompilieren: <code>linear-gradient(top bottom left right, ...)</code> (ohne Prefix ungültig) wird auf die Standard- <code>to ...</code> -Form gehoben.
2026-06	2.2.1	Handbuch-Aktualisierung (Versionshistorie ergänzt, Deckblatt-Version korrigiert); keine funktionalen Änderungen gegenüber 2.2.0.
2026-06	2.2.2	Zweisprachiges Handbuch (<code>microCSS-de.pdf</code> / <code>microCSS-en.pdf</code>) und zweisprachige READMEs (Englisch zuerst) für <code>gulp-mu-css</code> und <code>gulp-mu-ps</code> ; Build-Tooling für mehrsprachige Handbücher erweitert. Keine funktionalen Code-Änderungen.
2026-06	2.2.3	Fix: Überschriften tragen

Datum	Version	Anmerkungen
		explizite Outline-Level, damit das automatisch aktualisierte Inhaltsverzeichnis der PDF-Handbücher (DE/EN) befüllt wird (war zuvor leer). Reine Tooling-/Doku-Korrektur.

13 Rechtliches

Diese Software wird unter der MIT-Lizenz veröffentlicht und ist für freie und kommerzielle Nutzung freigegeben. Weder Dongleware noch der Autor haften für Schäden, die durch die Nutzung dieser Software entstehen. Die Nutzung erfolgt auf eigenes Risiko; sichern Sie Ihre Arbeit, bevor Sie die Werkzeuge einsetzen.

Autor: Meinolf Amekudzi.

13.1 MIT-Lizenz (Originaltext)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

13.2 Marken

Photoshop ist eine eingetragene Marke von Adobe Inc.; Affinity Photo ist eine Marke von Serif (Europe) Ltd. Die Nennung von Namen und Produkten dient ausschließlich der Information und stellt keinen Missbrauch der jeweiligen Handelsnamen oder Marken dar.

13.3 Verwendete Bibliotheken

μCSS und μPS nutzen Open-Source-Bibliotheken Dritter, insbesondere PostCSS (CSS-Parser, MIT-Lizenz), sharp (Bildverarbeitung, Apache-2.0-Lizenz) und ag-psd (PSD-Leser, MIT-Lizenz). Der Bin-Packer des Sprite-Atlas basiert auf node-bin-packing (©2011 Jake Gordon und Mitwirkende, MIT-Lizenz).