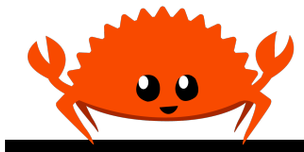


2D Graphics in Rust

Colin Rofls & Raph Levien

Rust *Obligatory Evangelism Slides

Rust *Obligatory Evangelism Slides

fast, reliable, productive: pick three

Rust fast, reliable, productive

Generates fast code: Minimal runtime, no GC, raw performance on par with C/C++

Lowest level: Direct access to graphics hardware & APIs

Enables performance-focused design: Safety guarantees make multi-threading easier*

*as in: easier to not mess up

Rust fast, reliable, productive

Ownership semantics: Memory is ‘owned’ by a particular scope. No shared mutable references (enforced by the compiler).

Rust fast, reliable, productive

```
fn main() {
    let mut my_vec = vec!['a', 'b', 'c'];

    for val in my_vec.iter() {
        if *val == 'b' {
            my_vec.insert(0, 'd');

        }
    }
}
```

```
error[E0502]: cannot borrow `my_vec` as mutable
because it is also borrowed as immutable

fn main() {
    let mut my_vec = vec!['a', 'b', 'c'];

    for val in my_vec.iter() {
               ------------ immutable borrow occurs here
        if *val == 'b' {
            my_vec.insert(0, 'd');
            ^^^^^^^^^^^^^^^^ mutable borrow occurs here
        }
    }
}

error: aborting due to previous error
```

Ownership* prevents many common memory errors.

**along with some other language features*

- dangling pointers/use after free and double free.

- buffer overflows: random access into a buffer is bounds checked

- null pointer dereferencing: (first class `Option` type)

Let the compiler track who is responsible for memory, not the programmer.

Rust fast, reliable, productive

Rust fast, reliable, productive

the expressiveness of a ‘high-level’ language.

Traits (interfaces) describe behaviour.

Rust productive: Traits

```rust
/// A generic trait for open and closed shapes.
pub trait Shape {

    /// Convert to a Bézier path.
    fn to_bez_path(&self, tolerance: f64) -> BezPath;

    /// Signed area.
    /// Only produces meaningful results with closed shapes.
    fn area(&self) -> f64;

    /// Total length of perimeter.
    fn perimeter(&self, accuracy: f64) -> f64;

    /// The smallest rectangle that encloses the shape.
    fn bounding_box(&self) -> Rect;

   ... some fields omitted
}
```

Rust productive: Traits

```rust
/// A circle.
pub struct Circle {
    pub center: Vec2,
    pub radius: f64,
}
```

Rust productive: Traits

```
impl Shape for Circle {

    /// Convert to a Bézier path.
    fn to_bez_path(&self, tolerance: f64) -> BezPath { … }

    /// Signed area.
    fn area(&self) -> f64 {
        PI * self.radius.powi(2)
    }

    /// Total length of perimeter.
    fn perimeter(&self, _accuracy: f64) -> f64 {
        (2.0 * PI * self.radius).abs()
    }

    /// The smallest rectangle that encloses the shape.
    fn bounding_box(&self) -> Rect {
        let r = self.radius.abs();
        let (x, y) = self.center.into();
        Rect::new(x - r, y - r, x + r, y + r)
    }
}
```

Rust productive: Traits

Traits are the basis for Rust's powerful generics system:

```
// A method on some sort of 'canvas' type
fn fill(&mut self, shape: impl Shape, color: &Color);
```

Rust productive: Traits

Traits are the basis for Rust's powerful generics system:

```
// A method on some sort of 'canvas' type
fn fill(&mut self, shape: impl Shape, color: &Color);

let our_round_friend = Circle::new((12.0, -404.0), 3.4);
canvas.fill(&our_round_friend, &orange);
```

```
let our_pointy_friend = Rect::new(0., -6., 101., 3.4);
canvas.fill(&our_pointy_friend, &orange);
```

Rust productive: Traits

Traits are the basis for Rust's powerful generics system:

```
// A method on some sort of 'canvas' type
fn fill(&mut self, shape: impl Shape, color: &Color);

let our_round_friend = Circle::new((12.0, -404.0), 3.4);
fill_for_Circle(canvas, &our_round_friend, &orange);
```

```
let our_pointy_friend = Rect::new(0., -6., 101., 3.4);
fill_for_Rect(canvas, &our_pointy_friend, &orange);
```

APIs like `Iterator` provide high-level abstractions that compile down to excellent code.

Rust productive: Iterator

```rust
/// A generic trait for open and closed shapes.
pub trait Shape {

    /// Convert to a Bézier path.
    fn to_bez_path(&self, tolerance: f64) -> BezPath;

}
```

Rust productive: Iterator

```rust
/// A generic trait for open and closed shapes.
pub trait Shape {
    /// The iterator resulting from `to_bez_path`.
    type BezPathIter: Iterator<Item = PathEl>;

    /// Convert to a Bézier path.
    fn to_bez_path(&self, tolerance: f64) -> Self::BezPathIter;
}
```

Rust productive: Iterator

```
pub struct CirclePathIter { … }

impl Shape for Circle {
    type BezPathIter = CirclePathIter;
    fn to_bez_path(&self, tolerance: f64) -> CirclePathIter { … }
}
```

Rust productive: Iterator

```rust
pub struct CirclePathIter { … }

impl Shape for Circle {
    type BezPathIter = CirclePathIter;
    fn to_bez_path(&self, tolerance: f64) -> CirclePathIter { … }
}
```

```rust
pub struct RectPathIter { … }

impl Shape for Rect {
    type BezPathIter = RectPathIter;
    fn to_bez_path(&self, tolerance: f64) -> RectPathIter { … }
}
```

Rust productive: Iterator

```
pub struct RectPathIter {
    rect: Rect,
    ix: usize,
}
```

Rust productive: Iterator

```
pub struct RectPathIter {
    rect: Rect,
    ix: usize,
}

impl Iterator for RectPathIter {
    type Item = PathEl;

    fn next(&mut self) -> Option<PathEl> {
        self.ix += 1;
        let Rect { x0, x1, y0, y1 } = self.rect;
        match self.ix {
            1 => Some(PathEl::Moveto(Vec2::new(x0, y0))),
            2 => Some(PathEl::Lineto(Vec2::new(x1, y0))),
            3 => Some(PathEl::Lineto(Vec2::new(x1, y1))),
            4 => Some(PathEl::Lineto(Vec2::new(x0, y1))),
            5 => Some(PathEl::Closepath),
            _ => None,
        }
    }
}
```

This gives us the abstraction of a high-level language (`fill()` instead of `fill_rect()`, `fill_path()`, etc) while still giving us fine grained control over the implementation.

Rust fast, reliable, productive, fun

Rust fast, reliable, productive, fun

Rust is a pleasure to use.

Rust fast, reliable, productive, fun

Friendly, helpful & inclusive community

Rust

Piet

Piet

(& Kurbo)

Piet

(& Kurbo)

(& Skribo, & Druid)

A 2D graphics abstraction layer

Piet

(& Kurbo)

(& Skribo, & Druid)

A 2D graphics abstraction layer

Piet

Curves & vector paths

(& Kurbo)

(& Skribo, & Druid)

A 2D graphics abstraction layer

Piet

Curves & vector paths

(& Kurbo)

(& Skribo, & Druid)

text layout & font handling

A 2D graphics abstraction layer

Piet

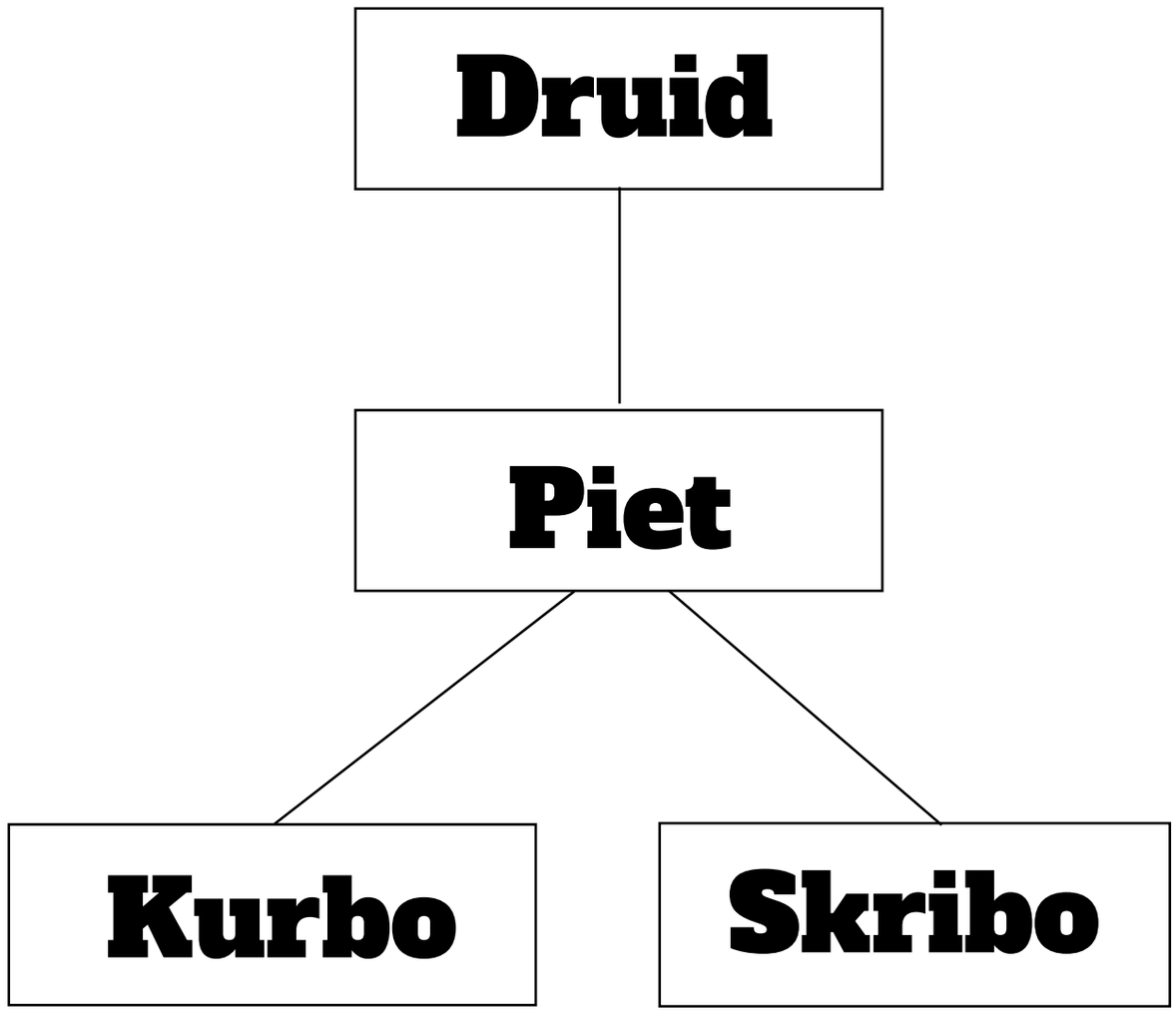
Curves & vector paths

(& Kurbo)

(& Skribo, & Druid)

text layout & font handling

A cross-platform gui toolkit


Druid
Piet
Kurbo
Skribo

Piet

Cross-platform

Piet

Piet defines a general set of traits that are then implemented for each target platform.

Piet

piet-cairo

piet-direct2d

piet-web

piet-cairo

piet-direct2d

piet-web

piet-metal

Piet-metal

Piet-metal

An experimental GPU renderer

Piet-metal

Fast

Piet-metal

Easier to fine-tune

Piet-metal

light on CPU

Piet-metal

Is traditional 2d API still valid? Or should UI be direct to GPU?

Piet-metal

How it works

Piet-metal

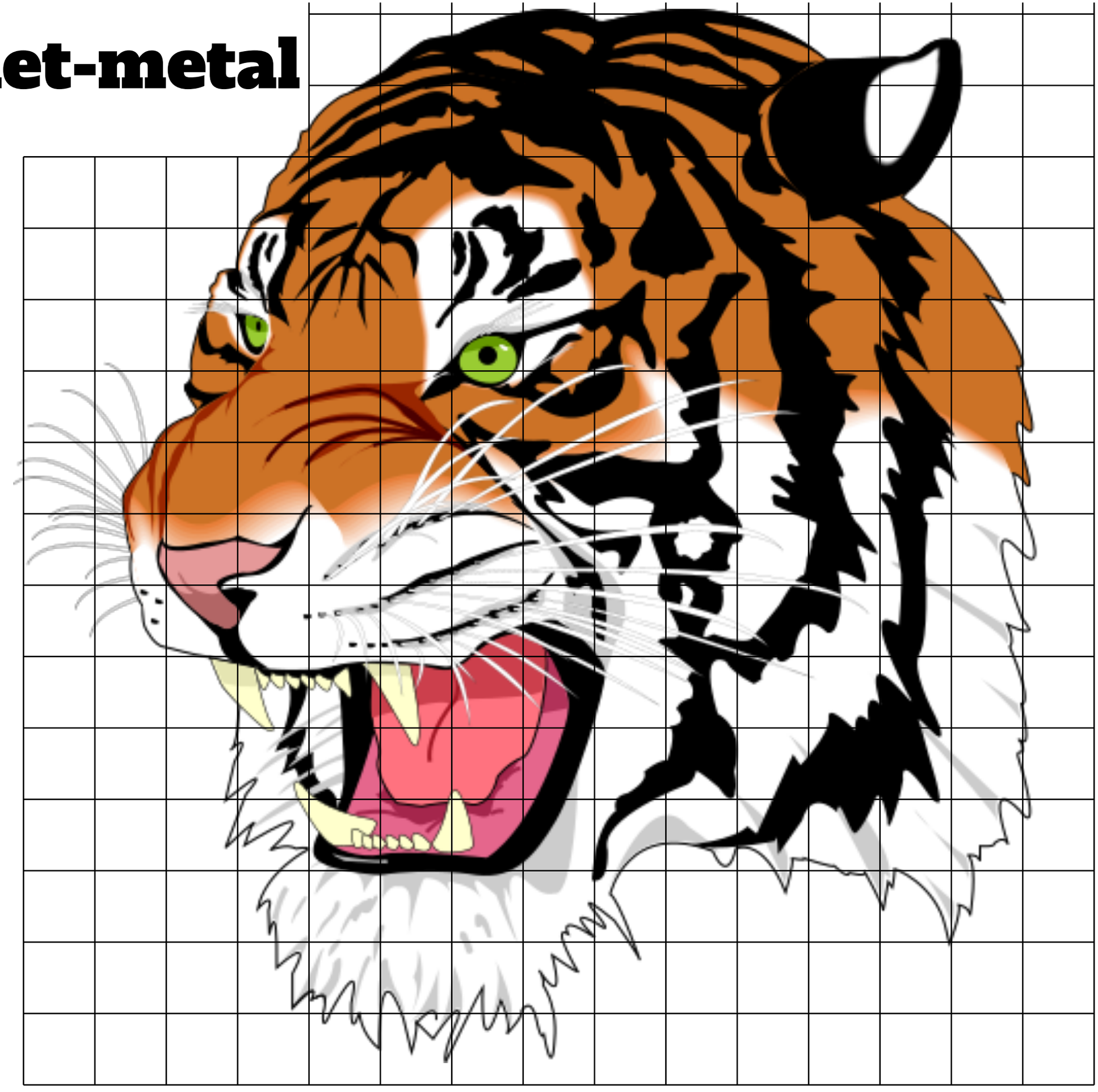
Two compute shaders

- First splits vector image into tiles
- Second renders each tile

I'm thinking some images here (probably split into multiple slides). The first is tiger with a square grid over it. The second is representation of a single tile.

Piet-metal

Piet-metal



Piet-metal

Piet-metal

*ironically not an SVG

Piet-metal

source vector image

per-tile command list

- solid(white)
- circle(green)
- circle(black)
- fill_edge(…)
- draw_fill(gray)
- stroke(black)
- …
- end

Piet-metal

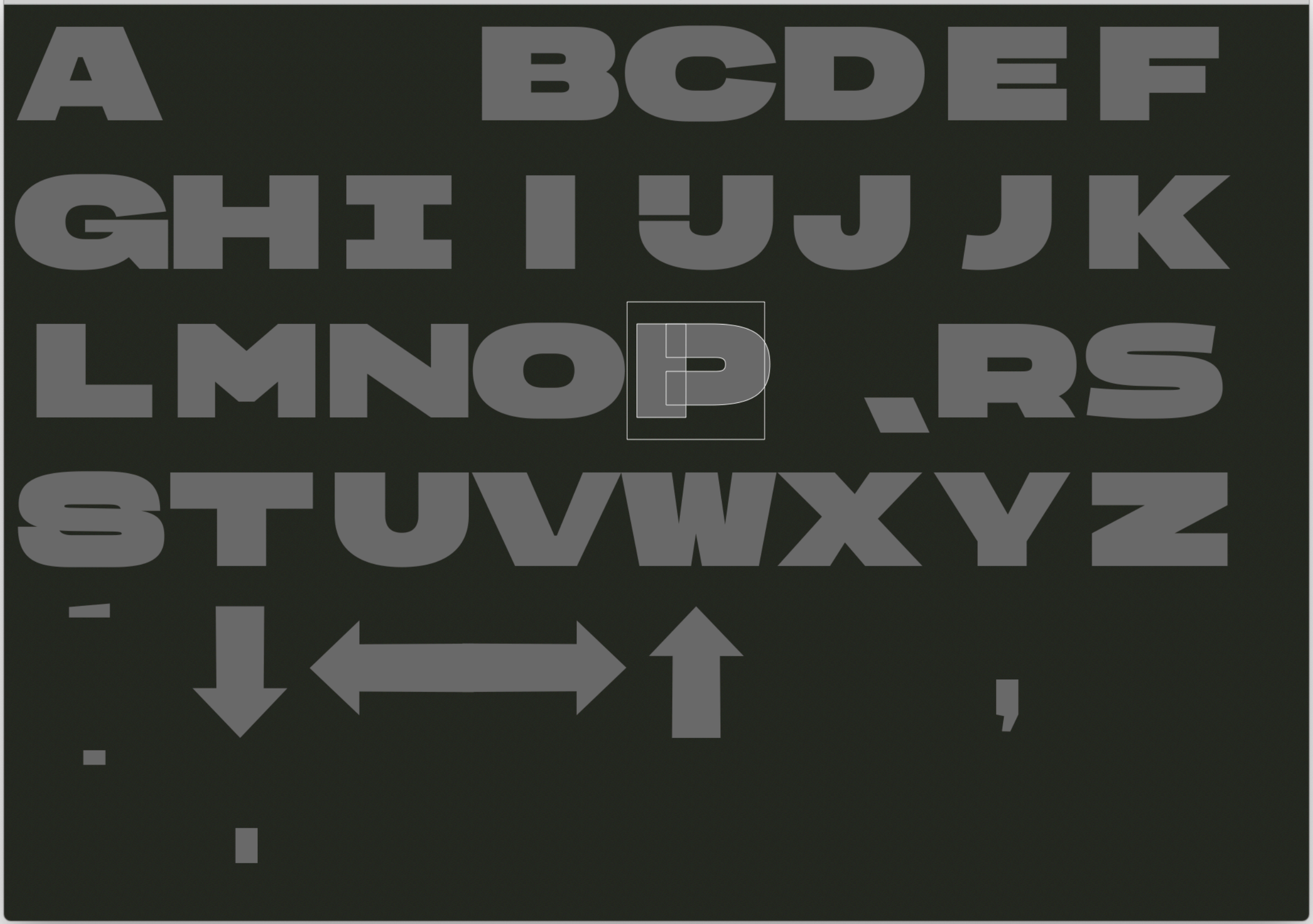
still a prototype,
but promising results

Piet

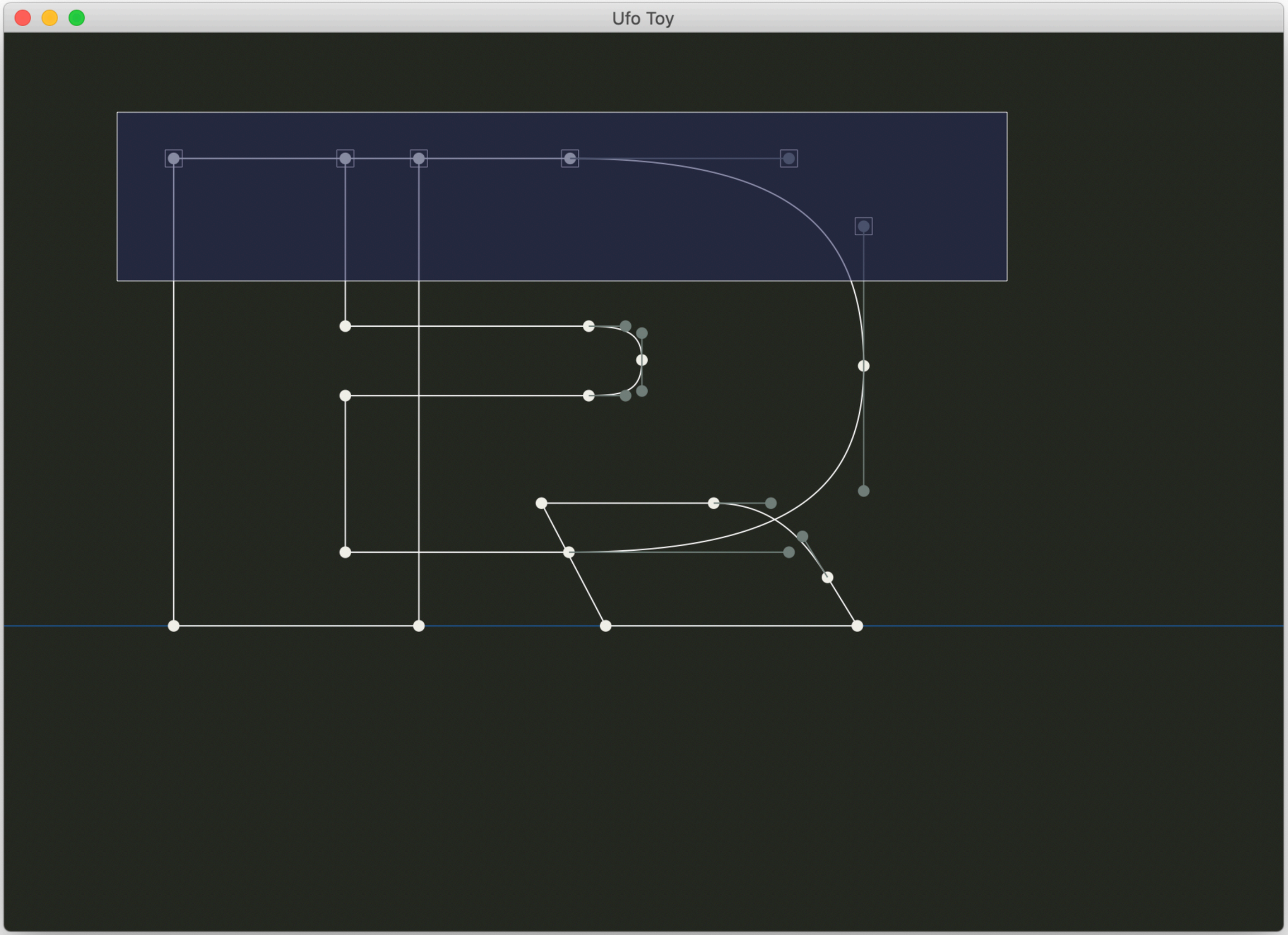
Demo

Piet

Ufo Toy



Piet



Piet

Thanks!

github.com/linebender

github.com/xi-editor/druid

xi.zulipchat.com

@raphlinus

@cmyr