Probabilistic Road Marking Detection using Algebraic Effects

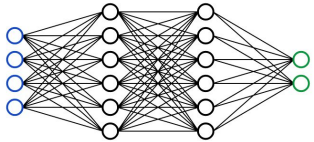
Bartłomiej Cieślar, Oliver Killane, Ethan Range,
Charlie Lidbury, Jordan Hall and Robert Buxton

GHOST

Labelling Training Data

~~Human Labellers~~

GHOST



Our Project

- **Slow**, 30 minutes per frame
- **Unsupervised** labelling

Ghost's detection AI

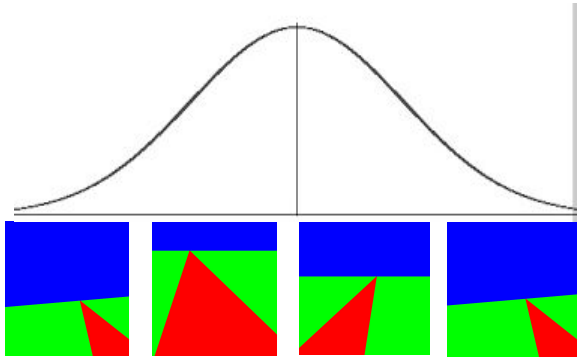
- **Fast**, real-time model
- Requires labelled data

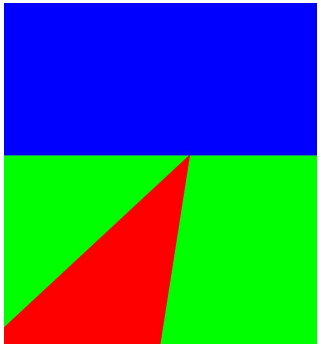
Labelling with Probabilistic Programming

How can we label data without supervision?

Input image

Probabilistic distribution of scenes

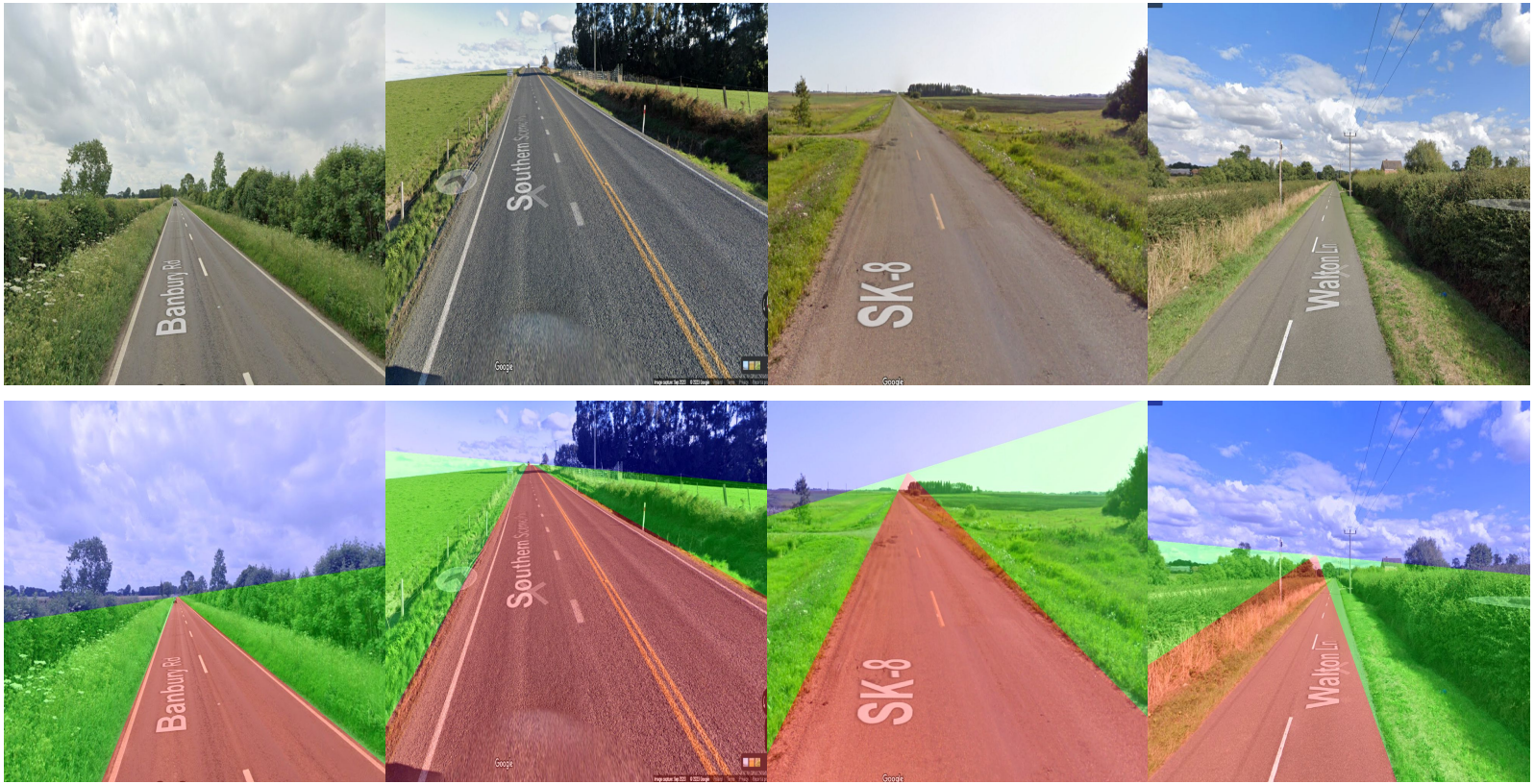




Scene most similar to input image

Putting detection into practice…

Running for longer...



Probabilistic Programming Languages of old...

- Separation between business logic and probabilistic code
- Lack of multi-modality
- Lack of modularity
- Lack of extensibility

Stan

Ghost requires:

- Easy extensibility
- Easy modularity
- **Flexibility**

PYRO

Anglican

… and of new

- ProbFX (2022)
- Haskell embedded DSL using **algebraic effects**
- Solves many aforementioned issues

Nick Wu

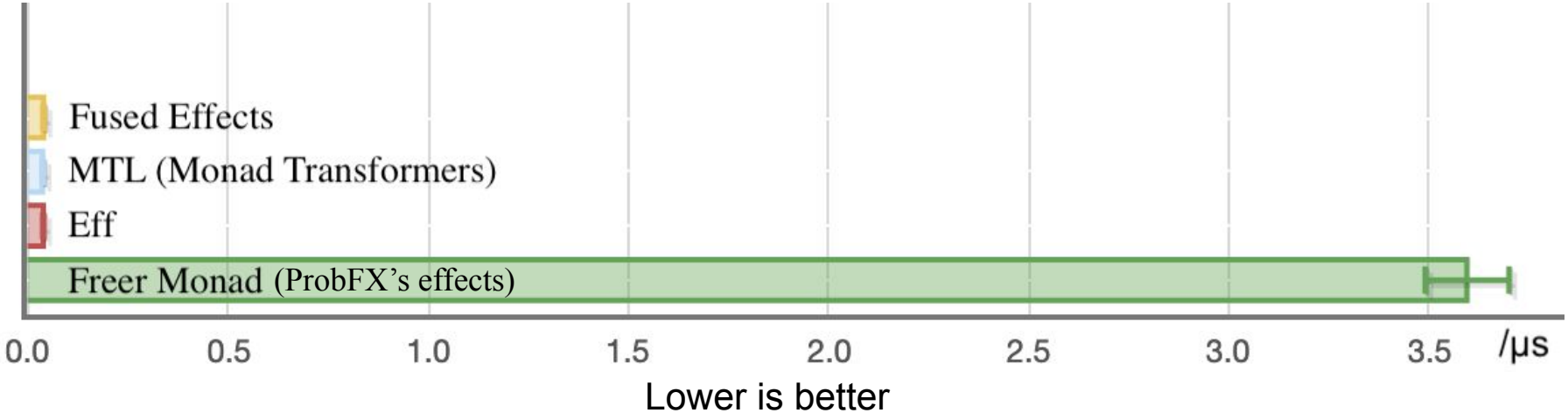
Min Nguyen

Designed for research, not production use

Introducing `fused-probfx`

Fused ProbFX

- `fused-effects`, an efficient library for effects based on Nick's work
- A re-write of ProbFX built on the `fused-effects` library
- Improved interoperability and performance potential



ProbFX vs `fused-probfx`

Conversion to `fused-effects`

Before: Custom Freer Monad is a tree

```
data Prog es a where
  Val :: a -> Prog es a
  Op :: EffectSum es x -> (x -> Prog es a) -> Prog es a
```

Expensive traversal over tree every time effect is handled, at runtime

```
handleLift :: forall m w. Monad m => Prog '[Lift m] w -> m w
handleLift (Val x) = return x
handleLift (Op u q) = case prj u of
    Just (Lift m) -> m >>= handleLift . q
    Nothing -> error "Impossible: Nothing cannot occur"
```

After: `fused-effects` - This traversal is done at compile time

Conversion to `fused-effects`

After: Additional effects can be used within probabilistic code (e.g. IO)

```
modelEff :: (Has (Lift Sampler) sig m, Observables env ["x", "y", "z"] Double)
         => Model env sig m ()
modelEff = do
   x <- normal 0 1 #x
   y <- uniform (-1) 1 #y
   Model $ sendM $ liftS $ print x
   Model $ sendM $ liftS $ print y
   return ()
```

Introducing Transitions

- The inference algorithm works by taking guesses on random variables
- Transition model allow the user to direct those guesses
- So the user can tune the algorithm’s behaviour, improving performance

```
let trans :: Transitions ModelTrans
    trans = (#x := transModel 20) <:> (#y := transModel 1) <:> nil
```

Converting the environment to a product type:

Before: Environment represented as a custom tree data structure

```
data Env (env :: [Assign Symbol *]) where
 ENil  :: Env '[]
 ECons :: [a] -> Env env -> Env (x := a : env)
```

After: Environment represented as a product type

```
data EnvElem (e :: Assign Symbol *) where
 Elem :: [a] -> EnvElem (x := a)

type Env = WP.Product EnvElem
```

Converting the environment to a product type:

This allows for greater flexibility when constructing environments:

- Allows complex constraints on environments (e.g. an environment contains a subset of variables)

```
foo :: (WP.Contains sampled (ExtractVars env)) => …
```

- Unifies constructors for environments, transition models, etc. under one constructor

```
env   = (#x := []           ) <:> (#y := [0]          ) <:> nil
trans = (#x := transModel 20) <:> (#y := transModel 1) <:> nil
```

- Allows folding and mapping over environments, transition models etc. polymorphically

```
pfoldr :: (forall a. f a -> x -> x) -> x -> Product f as -> x
pfoldr   x Nil         = x
pfoldr f x (Cons a p) = f a $ pfoldr f x p
```

Road Marking Auto Labeller

$$model \triangleq \begin{cases} x & \sim U(-0.5, 0.5) \\ y & \sim U(0.05, 0.5) \\ z & \sim U(-0.5, 0.5) \\ pitch & \sim U(-0.2, 0.2) \\ yaw & \sim U(-0.2, 0.2) \\ roll & \sim U(-0.2, 0.2) \end{cases}$$

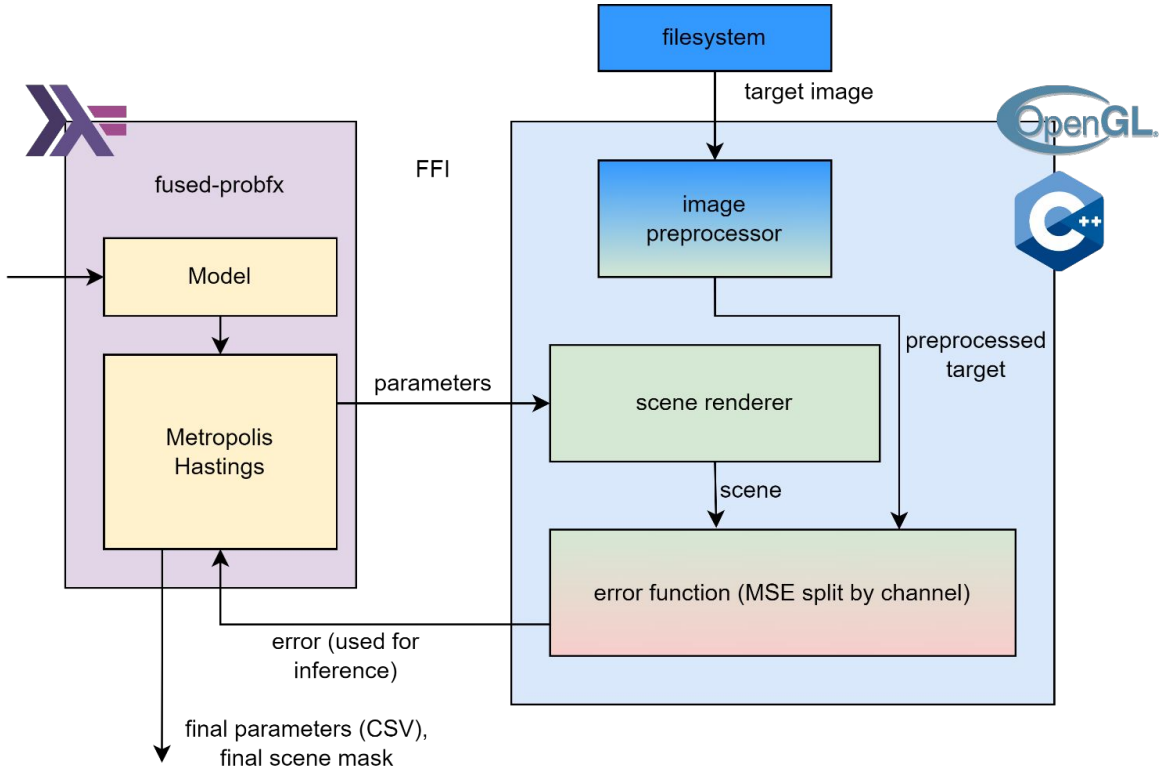
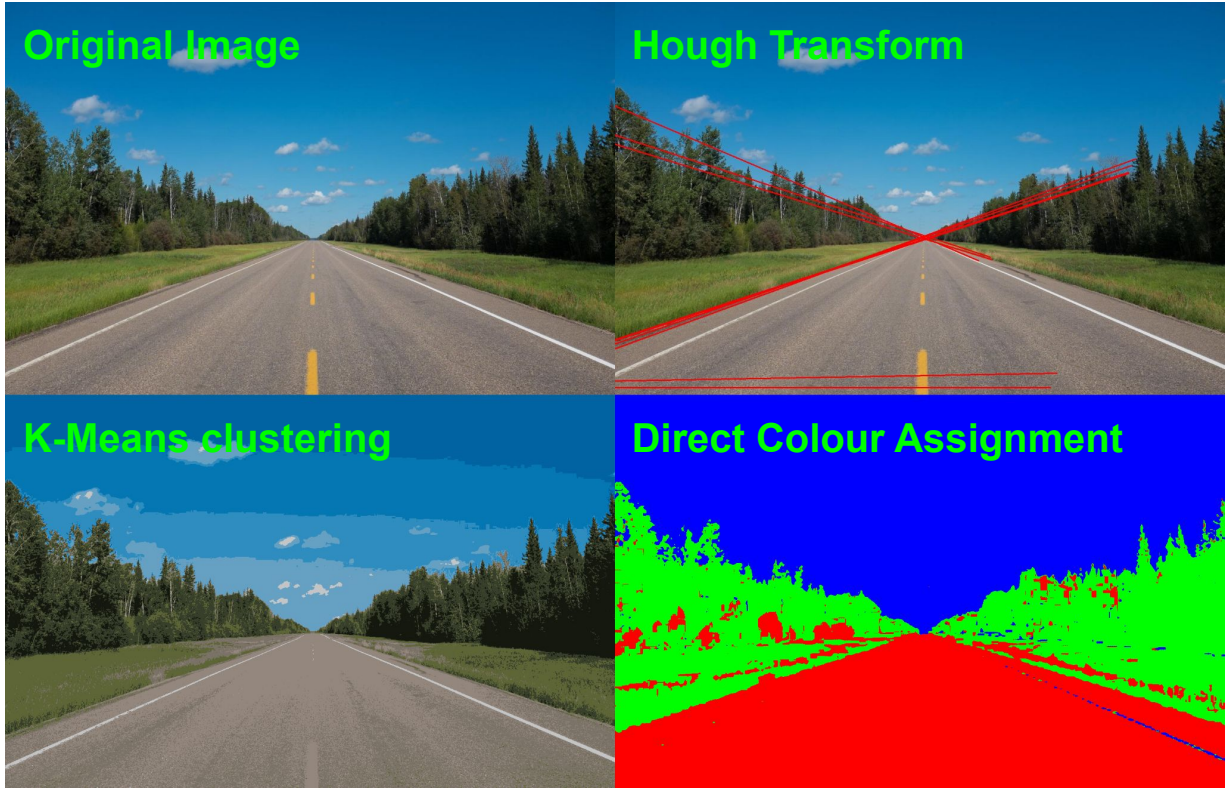
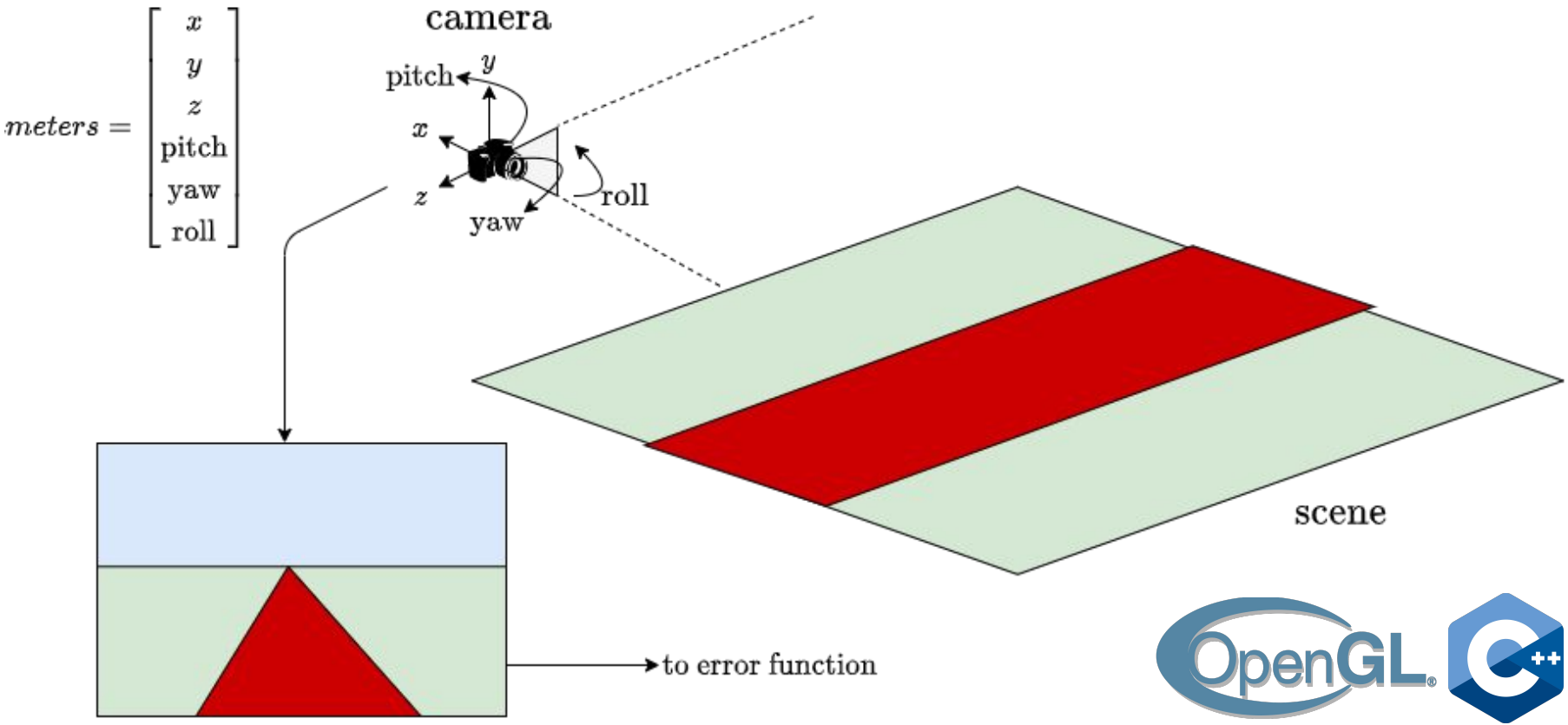


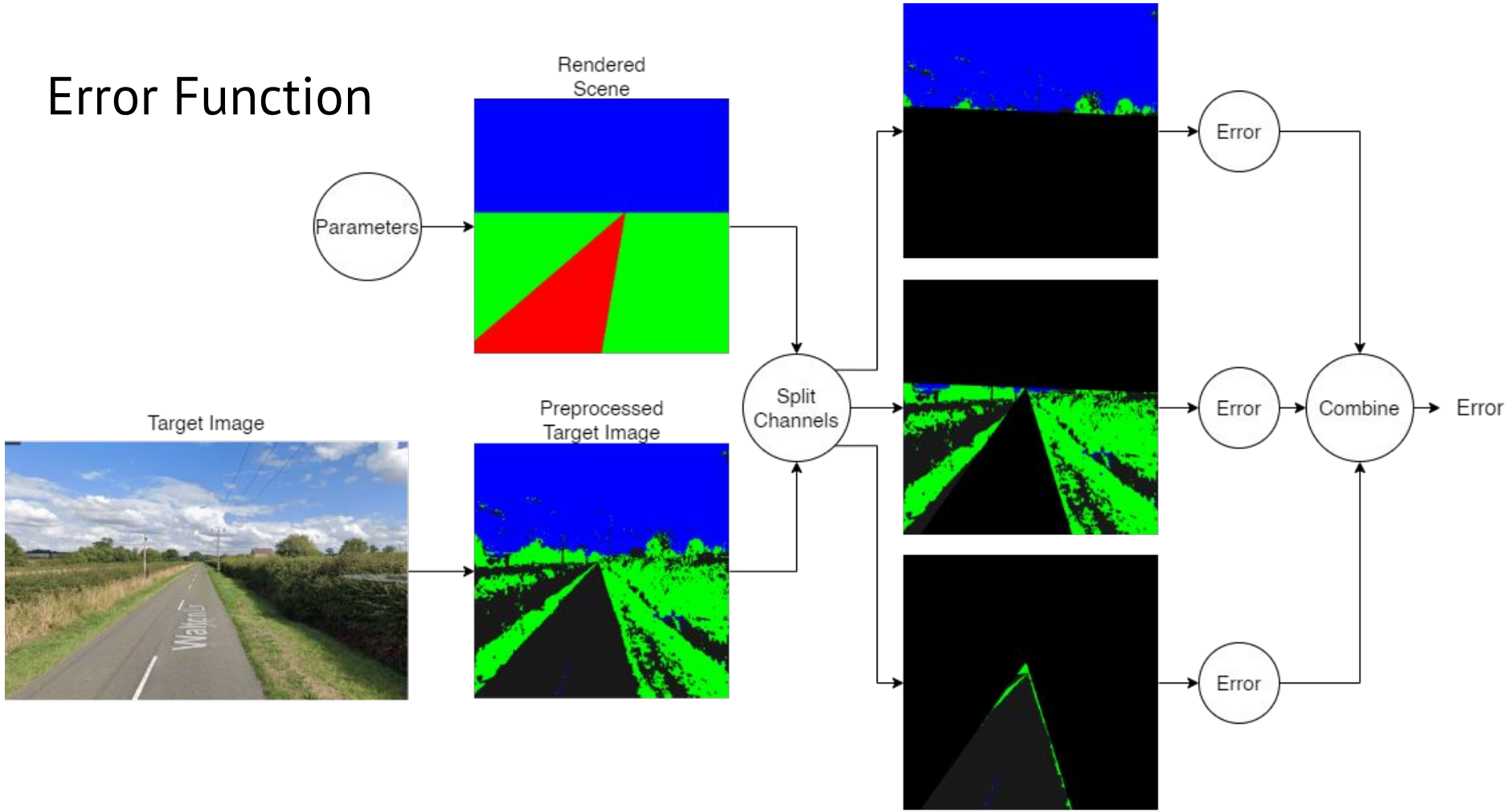
Image Pre-processing



Renderer


$meters = \begin{bmatrix} x \\ y \\ z \\ \text{pitch} \\ \text{yaw} \\ \text{roll} \end{bmatrix}$
camera
pitch
y
x
z
yaw
roll
scene
to error function

Error Function



General Evaluation

✓ Road markings provide excellent experimental test bed

- Arbitrary error function can be used
- Huge flexibility in model thanks to `fused-probfx`

✓ `fused-probfx` has great potential for interoperability and performance

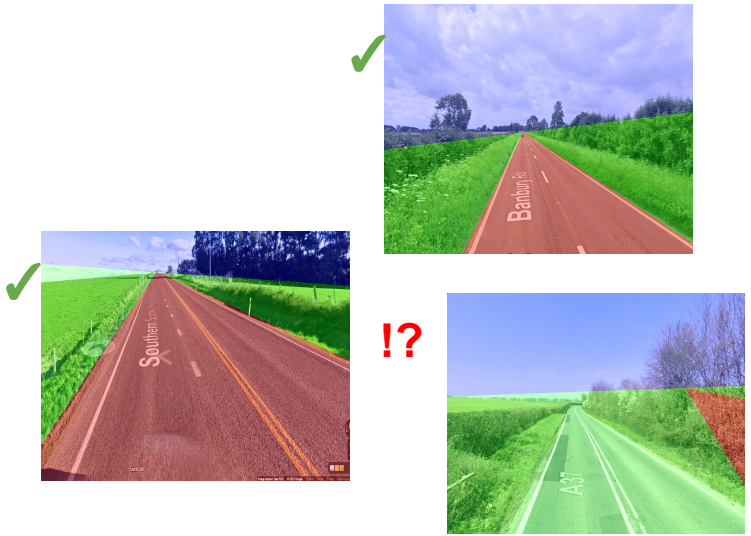
✘ Current performance of `fused-probfx` is worse than expected

✘ Lack of road detection on different road types / sceneries / times of day

Evaluation: Road markings detection

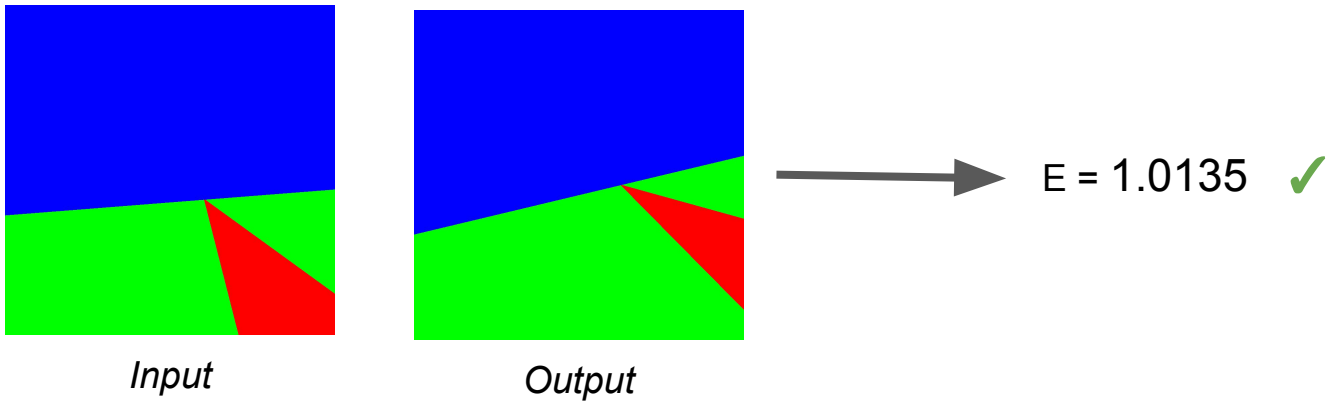
- Client meetings with Ben Lippmeier (Ghost Autonomy)

- Visual inspection with overlays



Evaluation: Road markings detection

- Synthetic benchmarks



Evaluation: `fused-probfx`

- Client meetings with Nicolas Wu
- Unit test suite ported from original ProbFX
- Benchmarks comparing to other PPLs