Retrocomputing with Clash

Haskell for FPGA Hardware Design

GERGŐ ÉRDI

https://unsafePerform.IO/retroclash/

8 Generative Graphics

The previous chapter ended with a way of generating a valid VGA signal, but without any content yet. Now it is time to try our hands at displaying something more interesting than a black screen. Remembering that `vgaDriver` provides the X and Y coordinates of the current pixel, and `vgaOut` takes an RGB triplet, the job here is to compute the color for each pixel based on its coordinates.

8.1 Combinational patterns

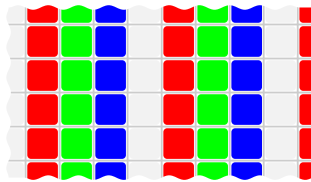
The most fundamental way of doing it is if we simply put a circuit between the coordinates and the color. For example, using a combinational circuit, we can draw red/green/blue/white stripes by looking at the bottommost two bits of the X coordinate. The logic itself is trivial:

```
rgbwBars
    :: (KnownNat w, KnownNat h, KnownNat r, KnownNat g, KnownNat b)
    => (Index w, Index h)
    -> (Unsigned r, Unsigned g, Unsigned b)
rgbwBars (x, y) = case fromIntegral x :: Unsigned 2 of
    0 -> red
    1 -> green
    2 -> blue
    3 -> white

black = (0, 0, 0)
red = (maxBound, 0, 0)
green = (0, maxBound, 0)
blue = (0, 0, maxBound)
white = (maxBound, maxBound, maxBound)
```

As the very polymorphic type of `rgbwBars` tells us, it can be used to drive a display at any resolution and at any color depth. Let's hook it up to our standard choice of $640 \times 480@60$ video mode:

```
topEntity
    :: Clock Dom25
    -> Reset Dom25
    -> VGAOut Dom25 8 8 8
topEntity = withEnableGen board
  where
    board = vgaOut vgaSync rgb
      where
        VGADriver{..} = vgaDriver vga640x480at60
        xy = liftA2 (,) <$> vgaX <*> vgaY
        rgb = maybe black rgbwBars <$> xy
```

Here, the type of `xy` is inferred to be `Signal _ (Maybe (Index 640, Index 480))`, which drives the instantiation of `rgbw` to `(w ~ 640, h ~ 480)`. Similarly, the type of `vgaOut` constrains the color depth to `(r ~ 8, g ~ 8, b ~ 8)`.



8.2 Stateful pattern generators

Our example function `rgbwBars` is nice and simple, but perhaps a bit *too* simple. For example, suppose we wanted to draw just red/green/blue stripes instead of red/green/blue/white. Calculating the modulus of the X coordinate by a power of 2 can be done by simply dropping the lowest bits, but here we would need to calculate it by 3 – not at all easy to do in a binary circuit.

Instead, we can use an RTL circuit to run a *counter* from 0 to 2, in lockstep with the X coordinate. This means our `rgbBars` will need to be a proper signal circuit, not just a pure function:

```
rgbBars1
    :: (KnownNat w, KnownNat h, KnownNat r, KnownNat g, KnownNat b)
    => (HiddenClockResetEnable dom)
    => Signal dom (Index w, Index h)
    -> Signal dom (Unsigned r, Unsigned g, Unsigned b)
```

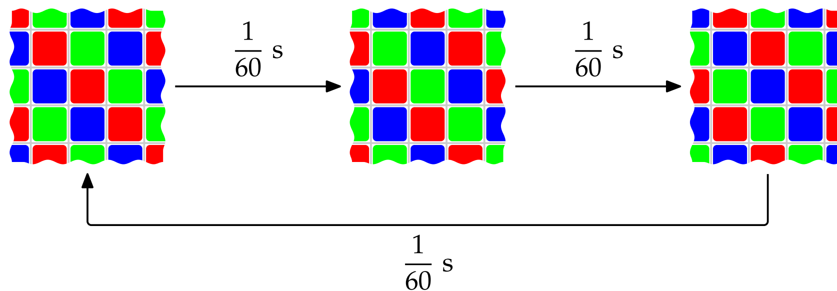
For a first try, we can simply increase an internal `Index 3` counter in every clock cycle, and use that as an index into a lookup table of colors:

```
rgbBars1 xy = colors !!. counter
  where
    counter = register (0 :: Index 3) $ nextIdx <$> counter

    colors = red :> green :> blue :> Nil
```

Hooking it up in `topEntity` is slightly different compared to the purely combinational `rgbBars`: since its input is now a `Signal` of coordinates, we have to always feed it something.

```
-- Inside topEntity
rgb =
    mux (isJust <$> xy) (rgbBars1 (fromMaybe (0,0) <$> xy)) $
    pure black
```

We're ready to try it out. However, once hooked up to a real screen, instead of nice vertical bars, we will see a checkerboard that flickers at 60 Hz:

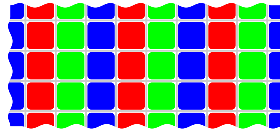


- The whole image is $800 \times 524 = 419,200$ clock cycles, not divisible by 3. This means every frame is drawn from a different starting state.

- Each line is 800 clock cycles, which is again not divisible by 3. Thus, every visible line is offset by 2 (the remainder of dividing 800 by 3) compared to the previous line.

We can fix both of these problems by restarting the counter at the start of each visible line. For this, we need to keep track of whether we are scanning the visible area: the corrected version of `rgbBars` gets the original, `Maybe`-wrapped coordinates, and if the X coordinate `isNothing`, resets the counter to 0. The rest of the implementation is unchanged:

```
rgbBars
    :: (KnownNat w, KnownNat h, KnownNat r, KnownNat g, KnownNat b)
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index w)
    -> Signal dom (Maybe (Index h))
    -> Signal dom (Unsigned r, Unsigned g, Unsigned b)
rgbBars x y = colors .!! counter
  where
    counter = register (0 :: Index 3) $
        mux (isNothing <$> x) (pure 0) (nextIdx <$> counter)

    colors = red :> green :> blue :> Nil
```



8.3 Animation

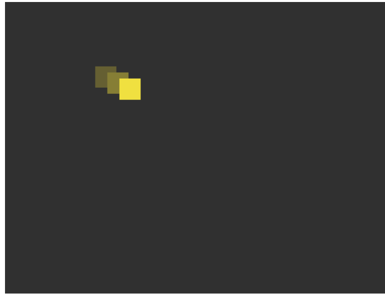
We have already, accidentally, implemented animated video in `rgbBars1`: since each frame started from a different state, the generated patterns were different frame by frame. For more controlled animation, we can keep an internal state describing the current frame, and synchronize its transition to the end of the frame.

We can detect the end of the frame simply by the vertical coordinate leaving the visible area. In the following circuit, each frame is rendered in a solid gray color, going from white to black. For simplicity's sake, we require all three color channel to have the same depth.

```
grayAnim
    :: (KnownNat w, KnownNat h, KnownNat c)
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index w))
    -> Signal dom (Maybe (Index h))
    -> Signal dom (Unsigned c, Unsigned c, Unsigned c)
grayAnim x y = bundle (brightness, brightness, brightness)
  where
    brightness = regEn 0 endFrame $ nextIdx <$> brightness
    endFrame = isFalling False (isJust <$> y)
```

Our second animated example will show a more involved state transition: we are going to draw a "ball" (more like a square) bouncing around, trapped between

the edges of the screen. Not only is this a simplified version of the bouncing DVD logo — a staple of early-2000s video players — but it should also give us inspiration for the next chapter, where we will build a fully playable Pong machine.



As before, our bouncing ball will simply be a pattern generator, i.e. a signal function from coordinates to current color:

```
type BallSize = 35

bouncingBall
    :: (KnownNat w, KnownNat h, KnownNat r, KnownNat g, KnownNat b)
    => ((BallSize + 2) <= w, (BallSize + 1) <= h)
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index w))
    -> Signal dom (Maybe (Index h))
    -> Signal dom (Unsigned c, Unsigned c, Unsigned c)
bouncingBall vgaX vgaY = draw
  where -- Continued below
```

The extra constraints on the minimal possible screen size are included because we are going to draw our ball as a 35×35 square, so we will need at least that much space; also, we will update the ball's position in each frame by a speed of $(2, 1)$, and our calculation would break down if there is not enough slack to move the ball by that much. At this point, this might look overly pedantic, since there are no VGA video modes with low enough resolution that this would be a problem. However, later in this chapter we will use this same circuit to demonstrate coordinate transformations which can create "virtual resolutions" internal to our circuit that don't actually exist at the video output.

The key to implementing `bouncingBall` is the bouncing logic, of course. Because all reflecting surfaces (the edges of the screen) are axis-aligned, we can save ourselves a heap of trouble by decomposing the ball's movement into a horizontal component, affected only by the vertical "walls", and a vertical component, bouncing between

the horizontal "walls". Thus, our state will be stored in two registers, updated at the end of each frame, each one bouncing between two endpoints:

```
frameEnd = isFalling False (isJust <$> vgaY)

(ballX, speedX) = unbundle $ regEn (0, 2) frameEnd $
    bounceBetween (0, rightWall) <$> bundle (ballX, speedX)
(ballY, speedY) = unbundle $ regEn (0, 1) frameEnd $
    bounceBetween (0, bottomWall) <$> bundle (ballY, speedY)
```

Before we give the definition of `rightWall` and `bottomWall`, we need to think about the type that we want to use for the state. For example, the X coordinate is given to us as a `Maybe (Index w)`, but it would be very painful to model all our ball dynamics using `Index w`; for example, a temporary value like `ballX + speedX` might not even be in bounds. Instead, we will do all `Index n` calculations in `Signed k` for a sufficiently large `k`. What is sufficiently large? Since `Index n` has possible values $0, 1, \ldots, n-1$, it takes up $\lceil \log_2 n \rceil$ bits. However, we want to use a signed representation for a more straightforward implementation of bouncing. Recall that `Signed k` is stored on `k` bits total, for a range of $-2^{k-1}, \ldots, 2^{k-1}-1$, so for a signed representation we need $\lceil \log_2 n \rceil + 1$ bits total:

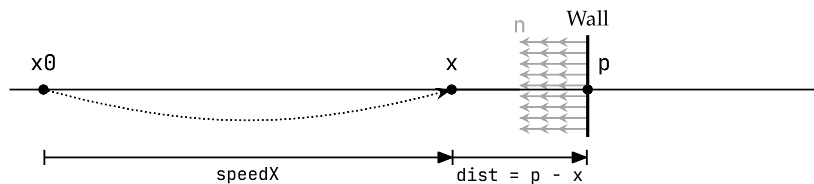
```
maxOf
    :: forall n p. (KnownNat n, 1 <= n)
    => p (Maybe (Index n)) -> Signed (CLog 2 n + 1)
maxOf _ = fromIntegral (maxBound :: Index n)

leftWall = maxOf vgaX - ballSize
bottomWall = maxOf vgaY - ballSize

ballSize :: (Num a) => a
ballSize = snatToNum (SNat @BallSize)
```

Given the current values of `ballX` and `ballY`, drawing is a simple matter of checking if `vgaX` and `vgaY` both fall within `ballSize` of them:

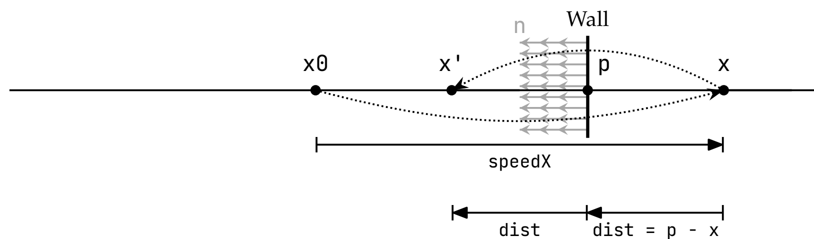
```
draw = mux isBall ballColor backColor

isBall = (near <$> ballX <*> vgaX) .&&. (near <$> ballY <*> vgaY)
  where
    near x0 = maybe False $ \(fromIntegral -> x) ->
        x0 <= x && x < (x0 + ballSize)

ballColor = pure (0xf0, 0xe0, 0x40) -- Yellow
backColor = pure (0x30, 0x30, 0x30) -- Dark gray
```

All that remains is implementing `bounceBetween` itself. There are many ways to do that; here we write a version that is based on multiple one-dimensional reflecting surfaces, which makes it easy to add additional "walls". We will use this ability in the full-fledged Pong game in the next chapter, to only include the player's paddle as a vertical reflection surface if it is at the same height as the ball.

Each reflecting surface is characterized with a point and a surface normal (a vector). If the ball is on the "far" side of the surface, then we need to mirror its position along the point. Remembering that we have decomposed our ball's motion into two one-dimensional components, the normal "vector" is one-dimensional as well, so instead of computing a dot product, we can simply compare signs to determine which side the ball is. If a reflection occurs, we adjust the position and negate the speed.



(1) `x` on the outside of `p` — No reflection



(2) `x` on the inside of `p` — Reflected to `x'`

```
reflect
    :: (Num a, Num a', Ord a, Ord a')
    => (a, a')
    -> (a, a')
    -> (a, a')
reflect (p, n) (x, dx)
    | sameDirection n dist = (p + dist, negate dx)
    | otherwise = (x, dx)
  where
    sameDirection u v = compare 0 u == compare 0 v
    dist = p - x
```

When bouncing between two walls, at each time step we apply the current speed to the position, and then reflect by two surfaces facing each other.

```
move :: (Num a) => (a, a) -> (a, a)
move (x, dx) = (x + dx, dx)

bounceBetween (lo, hi) = reflect (lo, 1) . reflect (hi, -1) . move
```

Exercises:

- Hook up some input switches to `bouncingBall` to independently change the horizontal and vertical speed of the ball. Of course, the direction of motion shouldn't change when the speed changes, only its magnitude.
- Draw a fixed-width border around the screen, and have the ball bounce between them
- Flash the screen for one frame whenever a bounce occurs. This will require changing `reflect` to report whether a reflection has occurred; we will need this functionality for Pong anyway.

8.4 Coordinate transformations

As we have seen, the basic operation of our video circuit is to take the current coordinates of the electron beam as input, and produce the desired color for that pixel as output. Conversely, by putting a coordinate transformer circuit in front, we can transform the output image.

For purely combinational circuits, any coordinate transformation works seamlessly. However, the situation is not that simple for stateful circuits. For example, `rgbBars`, as written, internally keeps a counter that is updated in every clock cycle inside the visible area. We can shift its image by feeding it a `Just` value for only a subset of the real visible area, and it would produce the correct output. However, if we tried to scale its output horizontally by feeding it the same X coordinate multiple times, the counter would be incrementing for each cycle just the same: the result is the same image as without scaling.

If, instead, we change `rgbBars` to only increment its counter when the current X coordinate is different from the previous one, we get a version that can be used in a wider variety of scenarios: any transformation (or composition of transformations) resulting in a monotonically increasing (i.e. left-to-right, top-to-bottom) signal of coordinates, when connected to this new `rgbBars`, would produce an image that is consistent with the transformation.

This scalable version of `rgbBars` is a bit tricky to get right. First off, we can compare `register Nothing x` with `x` to find out if `x` has changed in the current cycle. We would think that all that remains to do is change `register 0` in the definition of `counter` with `regEn 0 newColumn`, i.e. something like this:

```
scalableRGBBars1 x y = colors .!! counter
  where
    newColumn = register Nothing x ./=. x
    counter = regEn (0 :: Index 3) newColumn $
        mux (isNothing <$> x) (pure 0) (nextIdx <$> counter)
```

However, this version increments `counter` at the *start* of every "virtual" pixel. For example, if we scale horizontally by 4, i.e. if we use `scalableRGBBars1` with an `x` signal that changes value every 4th cycle, then the value of `counter` will be as follows:

Cycle	`x`	`newColumn`	`counter`	Output
0	`Nothing`	`False`	0	
1	`Nothing`	`False`	0	
2	`Just 0`	`True`	0	Red
3	`Just 0`	`False`	1	Green
4	`Just 0`	`False`	1	Green
5	`Just 0`	`False`	1	Green
6	`Just 1`	`True`	1	Green
7	`Just 1`	`False`	2	Blue
8	`Just 1`	`False`	2	Blue
9	`Just 1`	`False`	2	Blue
10	`Just 2`	`True`	2	Blue
11	`Just 2`	`False`	0	Red

If we started the register at `maxBound` instead of 0 when the visible area starts, it would take the first increment to set it to the desired value 0. This means while `counter` would still be wrong (it would lag one cycle behind the value we'd like), at least its next-value-to-be would be correct:

Cycle	`x`	`newColumn`	`counter`	`counterNext`
0	`Nothing`	`False`	2	2
1	`Nothing`	`False`	2	2
2	`Just 0`	`True`	2	0
3	`Just 0`	`False`	0	0
4	`Just 0`	`False`	0	0

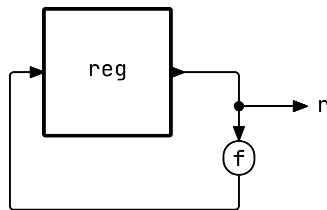
Cycle	x	newColumn	counter	counterNext
5	Just 0	False	0	0
6	Just 1	True	0	1
7	Just 1	False	1	1
8	Just 1	False	1	1
9	Just 1	False	1	1
10	Just 2	True	1	2
11	Just 2	False	2	2

Whenever we have a `register` whose value is lagging one cycle behind, we can solve that by tapping into its new value instead of the value propagated from the previous cycle. In general, this can be done by rewriting code of this form:

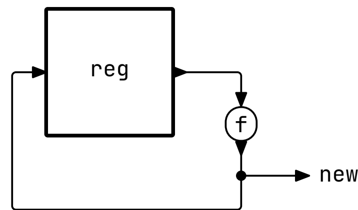
```
someCircuit1 = ... r ...
  where
    r = register x0 $ f r
```

into this:

```
someCircuit2 = ... new ...
  where
    r = register x0 new
    new = f r
```



(1) `someCircuit1`



(2) `someCircuit2`

This transformation, together with starting from `maxBound`, leaves us with the following version. Note that we can't use `regEn` here, because we need the new value `counterNext` to be already gated on `newColumn`. Thus the first `mux` in the definition of `counterNext`.

```
scalableRGBBars x y = colors .!! counterNext
  where
    newColumn = register Nothing x ./=. x
```

```
    counter = register (0 :: Index 3) counterNext
    counterNext =
        mux (not <$> newColumn) counter $
        mux (isNothing <$> x) (pure maxBound) $
        nextIdx <$> counter
```

All this is not to say that the original `rgbBars` was "wrong". Video pattern generators need to be designed for specific use cases, and if we are building a circuit that will generate video at the native resolution, it is perfectly fine to build a video subsystem that exploits this fact. Moreover, supporting coordinate transformations is not an all-or-nothing deal, but a spectrum – for example, as we have seen, `rgbBars` works as-is for horizontal or vertical translations, but required some changes to work for rescaling. Here, we present these two generally useful coordinate transformation circuits: translation and scaling.

8.4.1 Restricting the visible area

We can restrict the physical visible area of $w \times h$ to a smaller $w' \times h'$ by keeping the values of the virtual X and Y coordinate signals at `Nothing` for some of the time that the real X and Y coordinates are already `Just` values. The generic form of this transformation masks out parts of the visible area on both sides:

```
maskSides
    :: (KnownNat n, KnownNat m, KnownNat k)
    => (HiddenClockResetEnable dom)
    => SNat k
    -> Signal dom (Maybe (Index (k + n + m)))
    -> Signal dom (Maybe (Index n))
maskSides k raw = transformed
  where -- Continued below
```

We implement `maskSides` by starting a `Maybe (Index n)` counter whenever the `raw` input equals `Just k`. To make `maskSides` compose nicely with other transformers, we also implement the same logic as `scalableRGBBars` to only increment the counter whenever the `raw` input `changed`:

```
    changed = register Nothing raw ./=. raw
    started = raw .== Just (snatToNum k)

    r = register Nothing transformed
    transformed =
        mux (not <$> changed) r $
        mux (isNothing <$> raw) (pure Nothing) $
        mux started (pure $ Just 0) $
```

```
        (succIdx =<<) <$> r
```

We can use `maskSides` to define simpler combinators that mask out only the `k` pixels in the beginning or the end of the visible area. We write `k + n` and `n + k` in the type signature to be evocative of which side we're masking out:

```
maskStart
    :: forall k n dom. (KnownNat n, KnownNat k)
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index (k + n)))
    -> Signal dom (Maybe (Index n))
maskStart = maskSides (SNat @k)

maskEnd
    :: forall k n dom. (KnownNat n, KnownNat k)
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index (n + k)))
    -> Signal dom (Maybe (Index n))
maskEnd = maskSides (SNat @0)
```

With a bit more type-level arithmetic, we can also `center` a smaller image on a larger one: here, the ``k ~ ((n0 - n) `Div` 2`` constraint is effectively a type-level `let` binding for the size of the padding needed (rounded down). Because of rounding, we don't, in general, have `n0 ~ k + n + k`; instead, we set `m` to be the remaining visible area. The Clash typechecker for arithmetic constraints is powerful enough to solve `n0 ~ (k + n + m)`. In fact, we've already used the solver's power in the definition of `maskSides`, where the size `m` of the trailing side is inferred from knowing `k`, `n` and `k + n + m`. [1]

```
center
    :: forall n n0 k m dom. (KnownNat n, KnownNat n0, KnownNat k,
    KnownNat m)
    => (k ~ ((n0 - n) `Div` 2), n0 ~ (k + n + m))
    => (HiddenClockResetEnable dom)
    => Signal dom (Maybe (Index n0))
    -> Signal dom (Maybe (Index n))
center = maskSides (SNat @k)
```

[1] Clash automatically enables a handful of typechecker plugins implementing painless `KnownNat` propagation and arithmetic solvers for type-level naturals. These plugins can also be used with GHC proper, by adding `GHC.TypeLits.KnownNat.Solver`, `GHC.TypeLits.Extra.Solver`, and `GHC.TypeLits.Normalise` to the list of typechecker plugins loaded.

8.4.2 Scaling

Scaling is very similar to a generalized version of `scalableRGBBars`: we keep an internal counter that is incremented for every changed raw coordinate, and every time we would "start drawing a new red bar", the output coordinate is incremented. We can keep doing this until the output coordinate hits its maximum value, after which we reset it to zero the next time we enter the visible area. We also return the "sub-pixel" coordinate for visible pixels.

```
scale
    :: forall n k dom. (KnownNat n, KnownNat k, 1 <= k)
    => (HiddenClockResetEnable dom)
    => SNat k
    -> Signal dom (Maybe (Index (n * k)))
    -> (Signal dom (Maybe (Index n)), Signal dom (Maybe (Index k)))
scale k raw = (scaledNext, enable (isJust <$> scaledNext) counterNext)
  where
    prev = register Nothing raw
    changed = raw ./=. prev

    counter = register 0 counterNext
    counterNext =
        mux (not <$> changed) counter $
        mux (isNothing <$> prev) (pure 0) $
        nextIdx <$> counter

    scaled = register Nothing scaledNext
    scaledNext =
        mux (not <$> changed) scaled $
        mux (counterNext .== 0) (maybe (Just 0) succIdx <$> scaled) $
        scaled
```

The order of type variables is carefully chosen so that we can write `scale @n` in cases where the input and the output types wouldn't be constrained otherwise; for example, this allows us to write `scale @n (SNat @k) . center` to transform n to $m \geq k * n$ by making each pixel n times larger, and then sufficient padding on both sides.

Exercises:

- Write a 2D version of `rgbBars`, i.e. something that shows 9 different colors in a repeating, 3×3 tile. Hint: the vertical divide-by-3 counter should only be incremented at the end of each scanline.

- Combine two or more pattern generators by showing them in different parts of the screen. A simplest version would be showing `rgbwBars` on one half and `rgbBars` on the other half; for a more interesting challenge, try to define multiple window-like rectangles on the screen, each showing a different pattern.
- Change the bouncing ball circuit to internally use a resolution of 300×200, and render that to a 640×480 VGA mode by scaling up by two and centering horizontally and vertically.
- The above change leaves a border 20 pixels wide on both sides and 40 pixels high on the top and the bottom. Render these areas in some distinctive color, without changing anything in the definition of `bouncingBalls`.

8.5 Animation, differently

One drawback of writing `bouncingBalls` in the above style is that the full circuit is described as a monolithic function mapping signals to signals; as a consequence, it can be arbitrarily stateful. In this section, we rewrite it in a more principled way that will lend itself to high-level simulation.

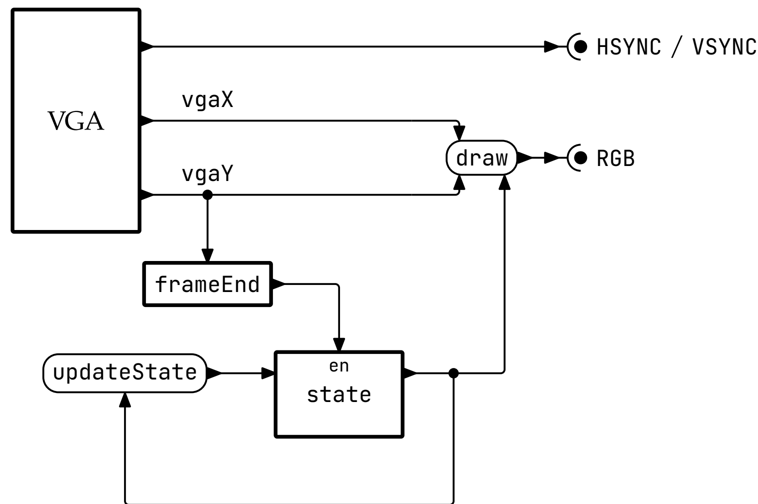
The basic idea behind the restructuring is to split `bouncingBalls` into two functions: a **state transition function** (the "bouncing") and a **drawing function** (the "balls"). Both of these are **pure** functions, with the state explicitly passed between them by the top-level circuit via a register. This is similar to the calculator from chapter 6.

In the calculator circuit, the state transition was triggered by user input. Animation, however, happens in real-time: the ball continues bouncing around on its own. We will use the start of the vertical blanking period to trigger for the state transition once for each frame, 60 times a second. This avoids potential graphical glitches that could occur if, for example, the ball position would be updated just at the same time as we are drawing its current position.

Moreover, although not a concern for our bouncing ball toy, for a more complicated machine it might take several cycles to compute a full state update; by starting just at the beginning of the vertical blanking period, we give us the most cycles possible before starting to draw the next frame.[2] For this reason, arcade machines and home computers generally have some way of signaling from the video subsystem to the CPU when the current frame is finished. In this computer-less circuit, we will simply consume the Y coordinate output of the video controller directly.

[2]Technically, we could start even a bit earlier, when the horizontal blanking period of the last line starts. The reason we aim for the start of the vertical blanking is to make the trigger logic simpler.

Putting it all together, our design will be as follows:



- A VGA controller generates the sync outputs and provides the rest of the circuit with the coordinates of the currently drawn pixel
- The animation state is stored in a register that is updated whenever the currently drawn frame ends.
- The drawing circuit takes the current state and the current video coordinates. By comparing the video coordinate to the ball's current position, it calculates the object at the current coordinate, which determines the color of the currently drawn pixel.

Since we have already written a VGA controller, the only parts we need to write are the following definitions:

- `data St`, the animation state.
- `updateState :: St -> St`, the state transition function.
- `draw :: St -> Index 640 -> Index 480 -> (Unsigned r, Unsigned g, Unsigned b)`, the drawing function.

As with the calculator, none of these parts use Clash `Signals` in their interface. We will exploit this property by assembling the same parts into a software implementation alongside the hardware one.

8.5.1 Animation state

Our only state is the ball position, stored in two pairs of `Coord`s to represent the ball's position and speed horizontally and vertically. We generate lenses here which we will use when writing `updateState`. The size of `Coord` is chosen to fit not only `ScreenWidth` and `ScreenHeight`, but also the intermediate calculations during `reflect`.

```
type Coord = Signed 10

data St = MkSt
    { _ballH, _ballV :: (Coord, Coord)
    }
    deriving (Show, Generic, NFDataX)
makeLenses ''St

initState :: St
initState = MkSt
    { _ballH = (10, 2)
    , _ballV = (100, 3)
    }
```

To future-proof our code somewhat, we also create a datatype for the animation's parameters; in this case, the only parameter is the ball size. This will be useful for one of the exercises later on.

```
data Params = MkParams
    { ballSize :: Coord
    }
    deriving (Show, Generic, NFDataX)

defaultParams :: Params
defaultParams = MkParams
    { ballSize = 35
    }
```

Since the bouncing ball is autonomous, there is no user input to `updateState`. We write it using the `State` monad so that we can compose `bounceBetween` for the vertical and the horizontal axis by `zooming` on the relevant fields of `St`. This is, arguably, overkill compared to something like `\(St x y) -> St (bounceBetween (0, w) x) (bounceBetween (0, h) y)`. However, when we move on to implementing more complex circuits, like a full-blown Pong game in the next chapter, we will need this flexibility for some of the stateful calculations like detecting the collision between the ball and the paddle. Note that we subtract

`ballSize` from the higher boundaries (bottom and right), since the coordinates stored in the state represent the ball's top-left corner.

```
updateState :: Params -> St -> St
updateState params@MkParams{..} = execState $ do
    zoom ballV $ modify $ bounceBetween (0, screenHeight - ballSize)
    zoom ballH $ modify $ bounceBetween (0, screenWidth - ballSize)
```

Although we could make the screen size a parameter, here we go a different route to ensure that the ball stays exactly in the visible area: `screenWidth` and `screenHeight` are reflected from type-level constants which we will also use in the type of `draw`:

```
type ScreenWidth = 640
type ScreenHeight = 480

screenWidth :: Coord
screenWidth = snatToNum (SNat @ScreenWidth)

screenHeight :: Coord
screenHeight = snatToNum (SNat @ScreenHeight)
```

8.5.2 Drawing

We implement drawing by writing a pure function that is only concerned with visible pixels. For the color channels, although the final output is limited in depth by the targeted hardware platform, here we use `Word8` for each channel for 24-bit colors. This allows us to specify the colors we'd like, not just the ones we can have; the latter can be derived trivially in `topEntity` by just truncating the lowermost bits. However, using `Word8` will make simulation performance dramatically better. That is because conversions like `bitCoerce :: Unsigned 8 -> Word8`, while a no-op in a real hardware circuit, involves a significant simulation overhead, so we are better off if we can avoid it three times for each pixel during simulation. By defining `Color` this way, we only have coercions in the other direction, `bitCoerce :: Word8 -> Unsigned 8`, in `topEntity` which is outside the context of our high-level simulation.

```
type Color = (Word8, Word8, Word8)

draw :: Params -> St -> Index ScreenWidth -> Index ScreenHeight -> Color
draw MkParams{..} MkSt{..} ix iy
    | isBall    = yellow
    | otherwise = gray
  where -- Continued below
```

This leaves us with only the problem of calculating if a given `(ix, iy)` coordinate is within the area where we want the ball to be visible. Later, more complete versions of `draw` will have exactly the same structure, just with more branches for `isWall` and `isPaddle`.

The workhorse function of `isBall` and similar definitions is determining if the current pixel is within a given axis-aligned rectangle. But first, we convert `ix` and `iy` to `Coords` to be compatible with `St`'s fields.

```
x = fromIntegral ix
y = fromIntegral iy

z `between` (lo, hi) = lo <= z && z <= hi
rect (x0, y0) (w, h) =
    x `between` (x0, x0 + w) &&
    y `between` (y0, y0 + h)
```

Given these definitions, we can write `isBall` simply by checking if the current pixel is in a `ballSize` × `ballSize` rectangle starting at the ball position:

```
(ballX, _) = _ballH
(ballY, _) = _ballV

isBall = rect (ballX, ballY) (ballSize, ballSize)
```

8.5.3 The top-level circuit

Let's take stock of the components we have written so far:

- `data St` is our state, which we want to keep in a register.
- `updateState` is the state transition function, which should be used to update the register at each `frameEnd`
- `draw` is the drawing function which calculates the color of the currently rendered, visible pixel

Components we need to assemble it into a full circuit:

- A VGA controller to generate the sync signals and to keep track of which pixel is currently rendered, if any.
- A signal `frameEnd :: Signal _ Bool` that fires at the end of each frame, to trigger the state register's update. This can be implemented by checking the Y coordinate output of the VGA controller ceasing to be `isJust`, since that means we have left the last visible line.

With these considerations, and using a 25.175 MHz clock as before for our chosen VGA mode, the full top-level circuit is as follows:

```
createDomain vSystem{vName="Dom25", vPeriod = hzToPeriod 25_175_000}

topEntity
    :: "CLK_25MHZ" ::: Clock Dom25
    -> "RESET"     ::: Reset Dom25
    -> "VGA"       ::: VGAOut Dom25 8 8 8
topEntity = withEnableGen $ vgaOut vgaSync rgb
  where
    VGADriver{..} = vgaDriver vga640x480at60
    frameEnd = isFalling False (isJust <$> vgaY)

    st = regEn initState frameEnd $ updateState defaultParams <$> st

    rgb = fmap (maybe (0, 0, 0) bitCoerce) $
        liftA2 <$> (draw defaultParams <$> st) <*> vgaX <*> vgaY
```

Since all our changes in this section so far were just shuffling around parts of the previous bouncing ball circuit, it shouldn't be a surprise that compiling, synthesizing, and uploading this circuit to an FPGA will result in the same video output as before. So, what was the point?

8.6 High-level simulation with SDL2

Restructuring our circuit into two separate, pure functions, one for the state transition and the other to implement drawing, pays dividends when we want to write a simulator for it. Just like in the calculator project, here we can exploit the structure of our code by assembling `updateState` and `draw` differently, into a sequential, stateful program that does the following:

0. Create a window with a backing texture of size 640×480.
1. Poll for any potential user input; exit if the window is closed.
2. Pass the current state and all possible coordinate pairs $(0, 0), (0, 1), \ldots, (639, 479)$ to `draw` and record its output into the texture.
3. Apply `updateState` to the state and the interpretation of the input events.
4. If we have some time left out of the $1/60$ of a second frame time, sleep for the remainder.
5. Repeat from step 1.

We are going to use *SDL2* to take care of the nitty-gritty of opening windows and polling keyboard events in whatever operating system we use. SDL2 is a mature

cross-platform library with good Haskell bindings that don't get in the way.

8.6.1 Hello, SDL2!

Before jumping head-first into connecting `draw` and `updateState` to SDL2, let's write a standalone program that uses SDL2 to open a window, draw some pixels, and waits for a keypress before shutting down. This will show us all we need to know about SDL2, and the rest will be up to us.

```
import SDL
import Data.Word

-- These are needed since textures are accessed through pointers
import Foreign.Ptr
import Foreign.Storable
```

We start `main` by initializing SDL and creating a `window`. The window size is scaled from the intended virtual screen size; we use a screen size with a resolution of just 24×18 so that we can easily see the result of drawing a single pixel. The window itself will have size 720×540.

```
screenSize = V2 24 18
screenScale = 30

main :: IO ()
main = do
    initializeAll
    window <- createWindow "Hello, SDL2!" defaultWindow
    windowSize window $= (screenScale *) <$> screenSize
    withTexture <- setupTexture window

    forever $ withTexture drawHello
  where -- Continued below
```

In `setupTexture`, we attach a `renderer` and a `texture` to the `window`. We want an efficient way of drawing individual pixels, and that is exactly what the texture gives us. Most of the parameters here are not important for us, and we use some sensible default values. As we will see, the `RGB888` texture format is perhaps not exactly what its name suggests, but still the closest to our needs. The `TextureAccessStreaming` argument ensures we can get direct read-write access to the pixel data underlying the texture.

Given this `renderer` and `texture`, we can ask SDL for a raw texture pointer and manipulate pixels through it using the `lockTexture` / `unlockTexture` API. The callback function `drawTo` is given a `Ptr ()` to the beginning of the texture, and an

`Int` which is the rowstride, i.e. the pointer difference (in bytes) between the location of two pixels that are vertical neighbors.

```
    setupTexture window = do
        renderer <- createRenderer window (-1) defaultRenderer
        texture <- createTexture renderer RGB888 TextureAccessStreaming
    screenSize

        return $ \drawToTexture -> do
            (ptr, stride) <- lockTexture texture Nothing
            drawToTexture ptr (fromIntegral stride)
            unlockTexture texture
            SDL.copy renderer texture Nothing Nothing
            present renderer
```

Now we can get creative in `drawHello`. This is the point where we need to understand the `RGB888` texture format. The name would suggest that it uses three bytes per pixel, storing red, green, blue, then next pixel's red, and so on. Instead, each pixel is stored in *four* bytes, in machine byte order, with the fourth one unused. Accordingly, on a little-endian machine, we can write a version of `drawHello` that sets a single pixel at $(10, 5)$ to red by accessing each color byte separately in reverse (blue-green-red) order:

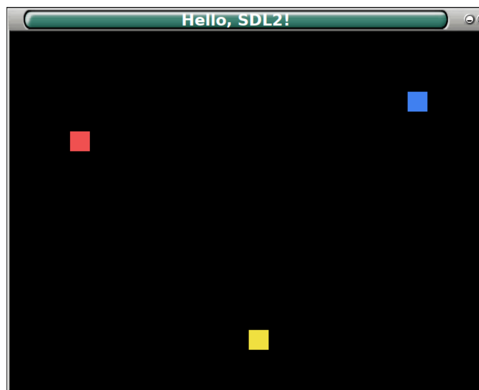
```
drawHello :: Ptr () -> Int -> IO ()
drawHello ptr rowstride = do
    pokeElemOff rowptr (x * 4 + 0) b
    pokeElemOff rowptr (x * 4 + 1) g
    pokeElemOff rowptr (x * 4 + 2) r
  where
    (x, y) = (10, 5)
    (r, g, b) = (0xf0, 0x50, 0x50) :: (Word8, Word8, Word8)
    rowptr = plusPtr ptr $ rowstride * y
```

This works, but it depends on the machine endiannness and in general just *feels awkward*. The reason for that is we are going against the grain here: SDL's intended texture access is via 32-bit values. If we change `drawHello` to write the color as a single `Word32`, not only does the code become cleaner, it will also have better performance once we move to changing more pixels than just one.

```
drawHello :: Ptr () -> Int -> IO ()
drawHello ptr rowstride = forM_ points $ \((x, y), rgb) -> do
    let rowptr = plusPtr ptr $ rowstride * y
    pokeElemOff rowptr x (rgb :: Word32)
  where
```

```
points =
    [ ((3,5),   red)
    , ((12,15), yellow)
    , ((20, 3), blue)
    ]

red    = 0xf0_50_50
yellow = 0xf0_e0_40
blue   = 0x40_80_f0
```

We can now run our program and marvel at the window showing our abstract art:



However, there is no nice way to exit our program; the only thing we can do is kill its process. This is because our main loop runs `forever`, with no way to exit. Instead of running it in `IO`, we will run it in `MaybeT IO`, so that we can exit the `forever` loop at any time by calling `mzero`. Luckily, SDL's API is already polymorphic over the base monad, so we don't have to wrap everything in `liftIO` calls.

Let's replace the main loop with something that only runs until we get an event from SDL that should prompt us to quit:

```
runMaybeT $ forever $ do
    events <- pollEvents
    keyDown <- getKeyboardState
    let shouldQuit =
            any isWindowCloseEvent events ||
            keyDown ScancodeEscape
    guard $ not shouldQuit
    liftIO $ withTexture drawHello32
```

Here, `shouldQuit` will be set to `True` if any of the latest `events` is a notification that the main window was closed, or if the Esc key is pressed.

```
isWindowCloseEvent ev = case eventPayload ev of
    WindowClosedEvent{} -> True
    _ -> False
```

This concludes our introduction to SDL2: we are now ready to connect our bouncing ball functions to texture drawing routines.

8.6.2 A reusable SDL2 simulator framework

Here we tweak our Hello World example to factor out the parts that will need to depend on the particulars of the simulated design:

- How the internal state is managed
- How input events `[Event]` and key states `Scancode -> Bool` are processed
- How the screen texture is updated

We will abstract over internal state management by allowing the simulation to happen in any `MonadIO m`. Input events and key states are going to be passed as simple function parameters.

For the screen texture update, we want to allow versatility to get good performance based on the access pattern (i.e. we want to expose the underlying texture directly), but also use types to track the screen size to rule out malformed indexing. We achieve this by making an abstract type `Rasterizer` indexed by the screen dimensions, and providing a library of trusted `Rasterizer` values for various use cases.

```
newtype Rasterizer (w :: Nat) (h :: Nat) = Rasterizer
    { runRasterizer :: Ptr () -> Int -> IO () }
```

Armed with this type, we can write a function that wraps the simulator in an environment where we keep running it as long as it returns the rasterizer for the current frame. This allows the simulator to decide to quit on its own. The only “free” parameters we have left are the window title and the screen scaling factor, since these are not inferrable from the types.

```
data VideoParams = MkVideoParams
    { windowTitle :: Text
    , screenScale :: CInt
    , screenRefreshRate :: Word32
    }
```

```
withMainWindow
    :: forall w h m. (KnownNat w, KnownNat h, MonadIO m)
    => VideoParams
    -> ([Event] -> (Scancode -> Bool) -> MaybeT m (Rasterizer w h))
    -> m ()
```

The implementation of `withMainWindow` is very similar to the Hello World program; we simply pass to `withTexture` a drawing function that runs the `Rasterizer` returned by the simulation.

```
withMainWindow MkVideoParams{..} runFrame = do
    initializeAll
    window <- createWindow windowTitle defaultWindow
    windowSize window $= fmap (screenScale *) screenSize

    withTexture <- setupTexture window
    let render rasterizer = withTexture $ \ptr rowstride ->
            liftIO $ runRasterizer rasterizer ptr rowstride

    runMaybeT $ forever $ do
        events <- pollEvents
        keyDown <- getKeyboardState
        let windowClosed = any isWindowCloseEvent events
        guard $ not windowClosed
        rasterizer <- runFrame events keyDown
        render rasterizer
    destroyWindow window
  where
    screenSize = V2 (snatToNum (SNat @w)) (snatToNum (SNat @h))

    setupTexture window = ... -- as before
```

Once the main loop finishes (because `windowClosed` or the simulation exits with `mzero`), we clean up the `window` by calling `destroyWindow`; this wasn't needed for our stand-alone Hello World program, since getting out of the the `forever` finishes the whole process anyway, cleaning everything up; but here we are building a library, so we have no control over whether clients will want to do other stuff after `withMainWindow` finishes.

There is nothing yet in our code that would lock it to 60 frames per second (or whatever else the refresh rate of a given circuit is). There's nothing we can easily do about it if our software simulation takes *more* than $1/60^{th}$ of a second, but if `runFrame` and `render`, taken together, take *less* time to run than the intended frame time, we can just sleep for the remaining frame time. To this end, we add the combinator

`atFrameRate`, which record the time (as measured, in milliseconds, by SDL's `tick` function) before and after the computation for a given frame. `waitFrame` then converts both the frame rate and the `before/after` time stamp into microseconds, suitable for `threadDelay`.

```
atFrameRate :: (MonadIO m) => Int -> m a -> m a
atFrameRate frameRate act = do
    before <- ticks
    x <- act
    after <- ticks
    waitFrame frameRate before after
    return x

waitFrame :: (MonadIO m) => Int -> Word32 -> Word32 -> m ()
waitFrame frameRate before after = when (slack > 0) $ liftIO $
    threadDelay slack
  where
    frameTime = 1_000_000 `div` frameRate
    elapsed = fromIntegral $ 1000 * (after - before)
    slack = frameTime - elapsed
```

Armed with `atFrameRate`, we simply replace our main loop's `forever $ do ...` with `forever $ atFrameRate screenRefreshRate $ do ...`.

We conclude our reusable simulator by writing a `Rasterizer` for combinational pattern generators: we iterate `y` through all possible values of `Index h`, calculate the pointer for each row using the row stride, and then poke the result of computing the pattern's color value starting from that pointer, 32-bit value by 32-bit value:

```
{-# INLINE packColor #-}
packColor :: Color -> Word32
packColor (r, g, b) =
    fromIntegral r `shiftL` 16 .|.
    fromIntegral g `shiftL` 8 .|.
    fromIntegral b `shiftL` 0

rasterizePattern
    :: (KnownNat w, KnownNat h)
    => (Index w -> Index h -> Color)
    -> Rasterizer w h
rasterizePattern draw = Rasterizer $ \ptr rowstride -> do
    for_ [minBound..maxBound] $ \y -> do
        let rowPtr = plusPtr ptr $ fromIntegral y * rowstride
        for_ [minBound .. maxBound] $ \x -> do
            pokeElemOff rowPtr (fromIntegral x) (packColor $ draw x y)
```

8.6.3 Let's see some bouncing balls finally!

Now that we've built up the infrastructure, it is time for the payoff: running our circuit design's `updateState` and `draw` functions and seeing their results on our screen in real time.

The idea is to pick `StateT St IO` as the monad we pass to `withMainWindow`. This takes care of holding on to the state from one frame to the next. Since `withMainWindow` wraps it in a `MaybeT`, we also have access to the effect of early termination, which we can use to implement a custom exit command. In this example, we will use the Esc key as an exit trigger.

To get back to vanilla `IO` for `main`, we simply use `evalStateT` to run the `StateT St IO ()` returned by `withMainWindow`. We have arranged the types of `updateState` and `draw` to minimize impedance mismatch with the `StateT` combinators and with `rasterizePattern`:

```
main :: IO ()
main =
    flip evalStateT initState $
    withMainWindow videoParams $ \events keyDown -> do
        guard $ not $ keyDown ScancodeEscape

        modify $ updateState defaultParams
        gets $ rasterizePattern . draw defaultParams
  where
    videoParams = MkVideoParams
        { windowTitle = "Bouncing Ball"
        , screenScale = 2
        , screenRefreshRate = 60
        }
```

And that's it!

Exercises:

- Tweakable parameters. Instead of passing the `defaultParams` to `updateState` and `draw`, change it into a proper signal. Connect some toggle switches as input to change the ball size mid-game.
- Extend the SDL simulator to generate the toggle switch state from keyboard events. For example, hook up the number keys 1 to 8 to each flip one virtual toggle switch.
- Similar to the earlier exercise, change this new version of the bouncing balls circuit to run in a virtual resolution of 300×200 pixels, scaled by two and

centered. The software simulation should render into a 300×200 window, and the hardware circuit will need to handle `Nothing` coordinates by drawing some nice border / background, passing `Just` the valid virtual coordinates to `draw`.

8.7 Summary

- A video pattern generator is a **potentially stateful circuit mapping coordinates to colors**. By default, the video controller's coordinate output is connected directly to the pattern generator's input, but we can put **coordinate transformers** between them.
- Just like in the calculator project, if we structure our design around a **state transition function** and a **pure output function**, this allows us to create a **high-level simulation** by hooking into the "interesting" parts of our design.
- For interactive, real-time designs with video output, such as video games, **sampling input and updating the state once per frame** is a natural and easily-implemented solution.
- The **SDL2** library provides an easy, robust and performant way of rendering pixel-based graphics, which can be used to **simulate video output in real time**.