Uzebox Mode 3 with Scrolling Guide

Matthew T. Pandina
artcfox@gmail.com

March 31, 2018

Introduction

The VRAM layout for the Uzebox video mode 3 with scrolling is topologically equivalent to a torus; it may be drawn as a flat square, but actually its right edge wraps around and connects to its left edge, and its bottom edge wraps around and connects to its top edge. The screen is a fixed size viewport able to display a portion of this VRAM. Conceptually, you can think of the screen or viewport as a square mapped somewhere onto the torus.

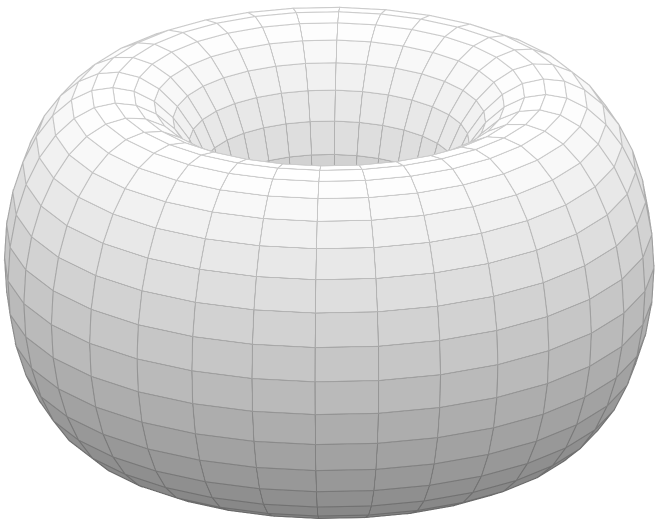


Figure 1: 32×32 VRAM layout, each square represents one 8×8 pixel tile

You can scroll the viewport in any direction by single pixel values, and it won't change the contents of VRAM, it only affects which pixel gets displayed in the top left corner of the screen. When you've scrolled the viewport a total of 256 pixels in a cardinal direction, the viewport will be in exactly the same position as it was before you started scrolling in that direction.

Since the viewport is smaller than VRAM, there will always be rows and columns that exist in VRAM, but aren't seen in the viewport. The goal when implementing scrolling in your Uzebox game is to load new tile data into those hidden portions of VRAM before the viewport scrolls those rows and columns into view.

As long as you keep loading new tile data into those hidden rows and columns (overwriting the old tile data that got scrolled off the other side of the screen), you can maintain the perception of a much larger world.

The diagrams in this document are intended to illustrate the relationship between the screen, VRAM, and the level tiles of your game, and will hopefully illustrate the nuances involved in populating VRAM with the proper tiles from your level array, based on the type of scrolling style you wish to implement.

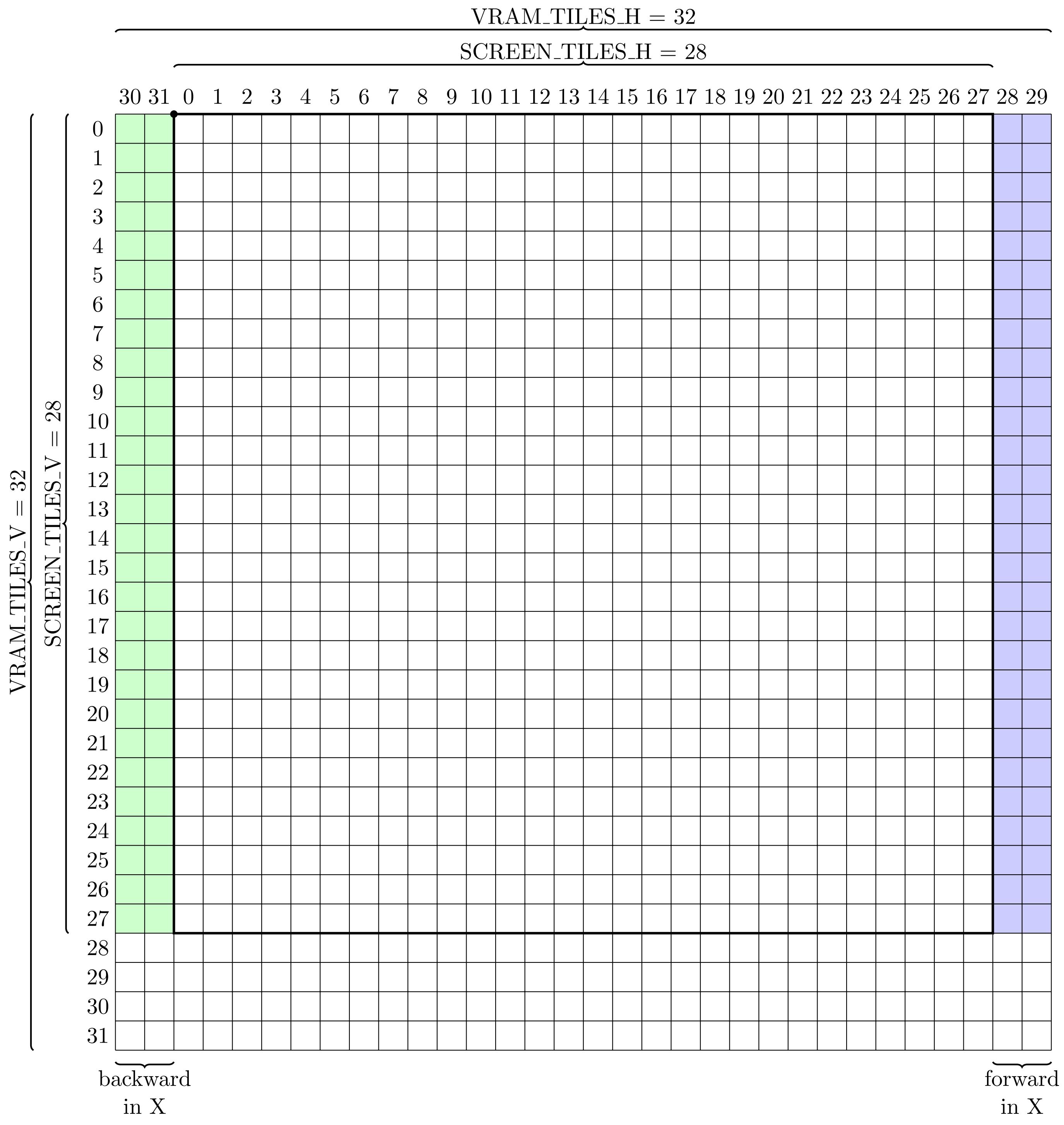
The latest version of this guide, along with complete versions of the projects described within, is available online.

Horizontal Scrolling

Conceptual Diagrams

The easiest way to visualize what level data needs to go where is to center the viewport in VRAM. Since we are limiting our discussion to just horizontal scrolling for now, the viewport is only centered horizontally. The dot represents the origin of VRAM and the origin of the screen. The viewport has not been scrolled at all, I just rotated the torus so the square is in the center, and then flattened it out.

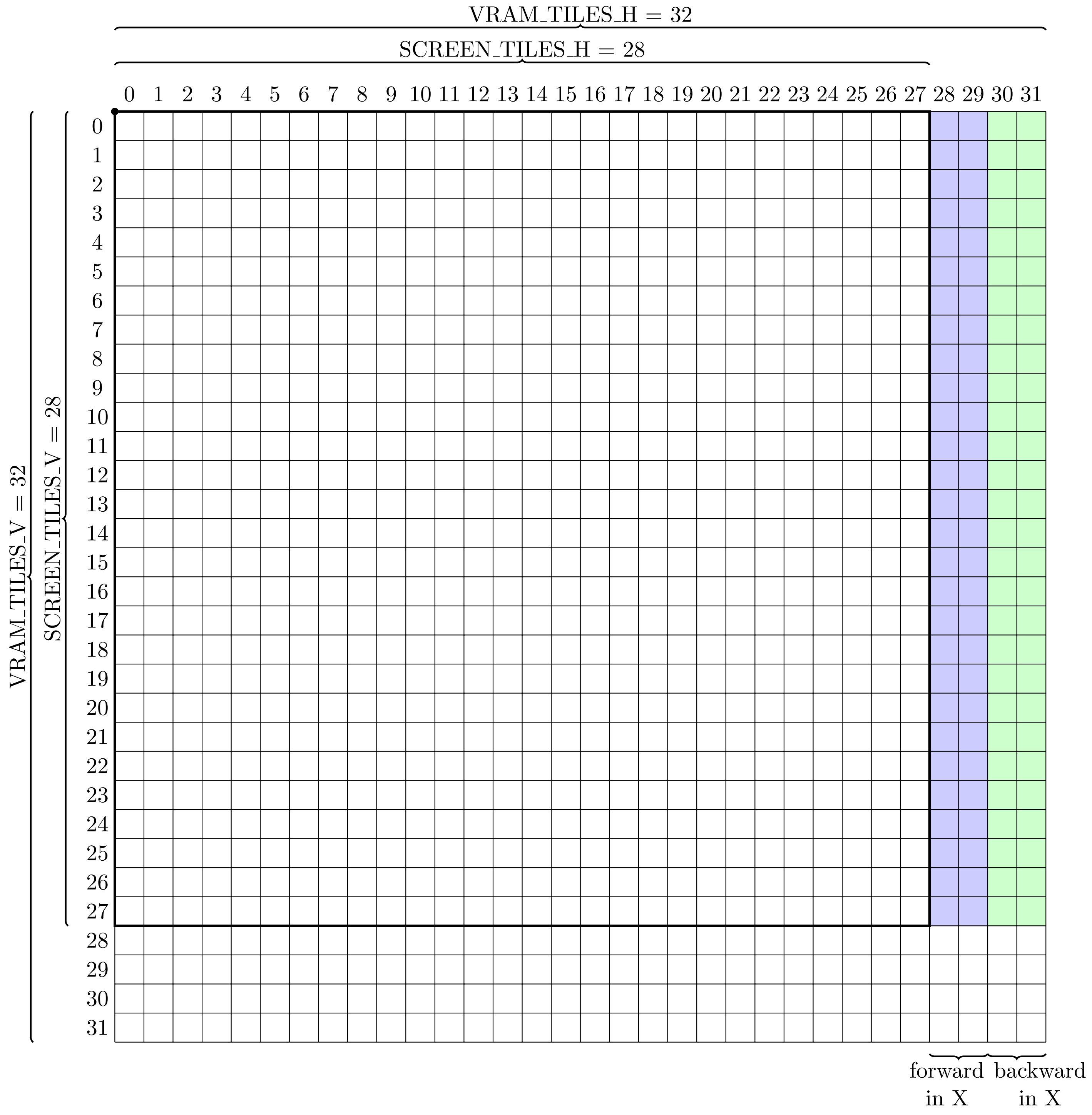
Since we are only scrolling horizontally, we can skip writing tile data to rows 28, 29, 30, and 31.



🟩 hidden level tiles backward in the X direction
🟦 hidden level tiles forward in the X direction

The diagram above was drawn with its viewport centered to make it easier for a person to conceptualize where the forward and backward level tiles need to go in VRAM, relative to the viewport. However, to populate the contents of VRAM programatically, it's useful to view the same diagram with the VRAM array wrapped so the origin appears in the top left corner, rather than wrapped to make everything symmetrical. Both views are equivalent, but this view should make it easier to see which tiles need to be populated using offsets and modulo arithmetic.

Keep in mind that when you are writing new tiles from your level data into hidden VRAM upon scrolling, you only need to write to the parts of hidden VRAM that are the farthest away from your viewport, which in this case would be columns 29 and 30. The other hidden columns will already be populated with the correct tiles.



🟩 hidden level tiles backward in the X direction

🟪 hidden level tiles forward in the X direction

Code

This code requires[1] your `Makefile` to have the following `KERNEL_OPTIONS` defined:

```
-DVIDEO_MODE=3 -DSCROLLING=1 -DSCREEN_TILES_V=28 -DSCREEN_TILES_H=28 -DVRAM_TILES_V=32
```

Very specific linker options are required for mode 3 with scrolling to function, so your `Makefile` must also include:

```
## Adjust the .data value to be 0x800100+VRAM_TILES_H*VRAM_TILES_V
LDFLAGS += -Wl,--section-start,.noinit=0x800100 -Wl,--section-start,.data=0x800500
```

I'm assuming you already have a Uzebox development environment installed and configured, and that you know how to create a Uzebox-specific tileset and define map variables in your `tileset.xml` file. If not, I created a video tutorial series that can walk you through this entire process.

The code in this section assumes your `tileset.xml` file looks like this:

```
<?xml version="1.0" ?>
<gfx-xform version="1">
  <input file="../data/tileset.png" type="png" tile-width="8" tile-height="8" />
  <output file="../data/tileset.inc" remove-duplicate-tiles="true">
    <tiles var-name="tileset"/>
    <maps pointers-size="8">
      <map var-name="map_horiz_level" left="0" top="0" width="60" height="28" />
    </maps>
  </output>
</gfx-xform>
```

Your map can have a different width, but its height should be 28 tiles, and the code assumes your map variable is called `map_horiz_level`. You are free to define additional levels, just give your new map variables unique names, and modify your code accordingly.

We will start with the list of `#include` files and the contents of the `main()` function, and then we can go back and fill in the rest. This should also give you a good overview of how the code will be structured.

Create a file called `horiz.c` using your favorite text editor, and write the following:

```
#include <stdint.h>
#include <stdbool.h>
#include <avr/io.h>
#include <stdlib.h>
#include <string.h>
#include <avr/pgmspace.h>
#include <uzebox.h>

#include "data/tileset.inc"

int main()
{
  SetTileTable(tileset);

  LEVEL l;
  CAMERA c;

  ClearVram();
  Level_initFromMap(&l, map_horiz_level);
```

[1]While `-DSCREEN_TILES_V=28` and `-DSCREEN_TILES_H=28` aren't strict requirements, they are the default values for mode 3 with scrolling, so for tutorial purposes that's what the code will assume. My philosophy is to start with the basics, get that working 100%, then refine, refactor, and make improvements. Once the initial code is written and proven to be correct, it'll be easier to spot the patterns, and then we can go back and make the code generic enough to handle different screen sizes. That is also the reason why this tutorial is not using any form of level compression. Solve one thing at a time, and you'll be able to isolate any bugs quicker.

```
  Camera_init(&c, &l);
  Camera_fillVram(&c);

  for (;;) {
    uint16_t held = ReadJoypad(0);
    if (held & BTN_LEFT)
      Camera_moveTo(&c, c.x - 1);
    else if (held & BTN_RIGHT)
      Camera_moveTo(&c, c.x + 1);

    Camera_update(&c);

    WaitVsync(1);
  }
}
```

As you can see, we have a `LEVEL` object and a `CAMERA` object, and each one has various functions associated with them. This allows us to keep everything organized, and it makes the code self-descriptive. Go ahead and read it line by line; you should get a pretty good idea of exactly what will happen.

The `LEVEL` object

Next we will define our `LEVEL` object. We want it to automatically keep track of the level's width, height, level data, and an offset into the level data. In this tutorial we will use the offset to skip the width and height that `gconvert` automatically adds to the beginning of our map data, but this offset can also be useful in a scenario where you have the data for multiple levels all packed together in a single array.

Below the `#include` lines, define the following struct:

```
typedef struct {
  uint16_t width;
  uint16_t height;
  uint16_t offset;
  const char *data;
} __attribute__ ((packed)) LEVEL;
```

The `__attribute__ ((packed))` is not strictly required, but it tells the compiler to not leave any padding between members of the struct.

Now define the `Level_initFromMap` function:

```
void Level_initFromMap(LEVEL *l, const char *map)
{
  l->width = pgm_read_byte(&map[0]);
  l->height = pgm_read_byte(&map[1]);
  l->offset = 2;
  l->data = map;
}
```

The first parameter is a pointer to the `LEVEL` object, and the second is a pointer to the level data. In our case, this will be the name of the map variable we defined in our `tileset.xml` file. Since that array data is actually stored inside the flash memory (PROGMEM) of the AVR rather than RAM, we need to use the `pgm_read_byte` function to access it.

In the future if you want to store your level data differently, i.e., not in a `gconvert` produced map, you can still use the `LEVEL` object, all you will need to do is add a different `Level_init` function that populates the `LEVEL` object appropriately.

Next, define a function that allows us to get a tile from the level based on an index into the data:

```
static inline uint8_t Level_getTileAt(LEVEL *l, uint16_t index)
{
  return pgm_read_byte(&l->data[l->offset + index]);
}
```

Make sure you define the function as `static inline` since it will be called from inside tight loops, and we don't want to incur the overhead of a function call every time around the loop. Again, we're using `pgm_read_byte` to access the level data, and we're adding the level offset we previously stored in the `LEVEL` object to the index given to us.

The last thing we need to complete our `LEVEL` object is to add the `Level_drawColumn` function:

```
void Level_drawColumn(LEVEL *l, uint8_t x, int16_t realX)
{
  if (realX < 0 || (uint16_t)realX > l->width - 1)
    return;

  uint8_t tx = x % VRAM_TILES_H;

  for (uint8_t i = 0; i < SCREEN_TILES_V; ++i) {
    uint16_t index = i * l->width + realX;
    SetTile(tx, i, Level_getTileAt(l, index));
  }
}
```

Here is where things start to get interesting, because we need to give it two x values. If you refer back to the conceptual diagrams, you'll see that `x` is needed to select which column of VRAM to populate, and `realX` is needed to select which column to read from in our level array in order to get the proper tile values. The variable `x` needs to wrap around the torus, but `realX` cannot because it is a reference to a specific column of level tiles.

This is also why `realX` is a signed data type, since it's possible for it to go negative when we are at the beginning edge of the level and trying to look backward.

The first line in the function ensures we don't read before the beginning of the level array when we are scrolled to the beginning of the level and try to look backward, and that we don't read past the end of the level array when we are scrolled to the end of the level and try to look forward.

To properly index into the VRAM array as if it were a torus, we need to use `x % VRAM_TILES_H` as the first parameter to `SetTile`, but we store it in the `tx` variable outside the loop so this calculation only has to happen once. This allows us to pass any `x` value we want into the `Level_drawColumn` function, and it will automatically be wrapped properly onto a valid VRAM column. Since this is the only place in the fuction we use `x`, we can get away with making its type `uint8_t`, because forcing 8-bit truncation won't change the result of the modulo by `VRAM_TILES_H`.

Next we loop over every visible row in the screen calculating an index into our level array, based on `realX` (which may be looking backward or forward) and call `SetTile` with the tile number returned from the `Level_getTileAt` function.

Notice that since we abstracted the concept of retrieving a tile from the level by calling the `Level_getTileAt` function, no changes to the `Level_drawColumn` function are required if, in the future, you decide to compress your level data. You still may want to make changes for performance reasons, but don't do that until you've fully debugged your level compression scheme!

The `CAMERA` object

Now that the `LEVEL` object is complete, we can define our `CAMERA` object. This one is so simple that you might wonder why we are even creating an object for it, but you'll appreciate the encapsulation later as you extend it and add features. Since we are only going to be scrolling horizontally, it just needs to keep track of the camera's x position in pixels across the width of the level, and store a pointer to the `LEVEL` object, so internally it can query the level and call its various functions.

Just below the `Level_drawColumn` function, define the following struct:

```
typedef struct {
  int16_t x;
  LEVEL *level;
} __attribute__ ((packed)) CAMERA;
```

Next, define the `Camera_init` function:

```
void Camera_init(CAMERA *c, LEVEL *l)
{
  c->level = l;
  c->x = 0;
  Screen.scrollX = (uint8_t)c->x;
  Screen.scrollY = 0;
}
```

Its implementation is self-explanatory; we store the level pointer, set the initial x position of the camera to zero, set the screen's x scrolling position to the camera's x position (zero), and set the screen's y scrolling position to zero to ensure it is in a known state.

As your game gets more developed you can imagine designing levels where your camera doesn't start at the far left of the level. A clean way to implement that would be to find a good place to store the initial x position of the camera for each level, have your customized `Level_init` function read it for that level (extend your `LEVEL` struct to store it), and finally query that value from your `LEVEL` object from within the `Camera_init` function, replacing the blind assignment to zero.

Now define a function we can use to set the camera's position:

```
void Camera_moveTo(CAMERA *c, int16_t x)
{
  if (x < 0) {
    c->x = 0;
  } else {
    int16_t xMax = (c->level->width - SCREEN_TILES_H) * TILE_WIDTH;
    if (x > xMax)
      c->x = xMax;
    else
      c->x = x;
  }
}
```

This function will automatically clamp any x value we call it with to a valid camera position for the currently loaded level. Remember that the camera's x position represents pixels across the level. For the assignment to `xMax`, the level width is stored in tiles, so we subtract the screen width (also in tiles) and then multiply the result by `TILE_WIDTH` to convert it into pixels. Because the origin of the viewport is at top left of the screen, the camera can't get any closer than a screen's width away from the right edge of the level.

Add the `Camera_fillVram` function next:

```
void Camera_fillVram(CAMERA *c)
{
  int16_t cxt = c->x / TILE_WIDTH;

  for (uint8_t i = 0; i < VRAM_TILES_H - 2; ++i)
    Level_drawColumn(c->level, cxt + i, cxt + i);

  Level_drawColumn(c->level, cxt + 30, cxt - 2);
  Level_drawColumn(c->level, cxt + 31, cxt - 1);
}
```

The first thing it does is convert the current camera's x position from pixels into tiles, and stores it in the variable cxt.

Next, refer to the second conceptual diagram to see which columns we can draw "normally," meaning by passing the same x and realX value to the Level_drawColumn function (hint: columns 0 through 29). Even though columns 28 and 29 are inside the hidden portion of VRAM, we can still index their VRAM column and the relative position of the tiles in the level array by adding one for each column beyond the camera's current position (represented by the dot in the diagram) that we move. That's why the for loop can go from 0 to VRAM_TILES_H - 2.

For the remaining two columns, we still need to increment their indices into VRAM (cxt + 30 and cxt + 31), but at the same time we need to reference columns from the level array that are backward in x, relative to the camera's current position, hence the cxt - 2 and cxt - 1.

If you are confused as to how much each of those last two columns needs to look backward in the level array, refer back to the first conceptual diagram where the viewport is centered, and match the column numbers up when moving to the second conceptual diagram. Now you understand why I drew both diagrams.

Also notice that we didn't have to deal with any VRAM wrapping, or worry about referencing positions outside the bounds of the level array here, because that's already been handled for us inside the Level_drawColumn function.

The only function we haven't defined yet is the Camera_update function which will scroll the screen, and draw the necessary column into the hidden portion of VRAM for us, so define that now:

```
void Camera_update(CAMERA *c)
{
  uint8_t prevX = Screen.scrollX;
  Screen.scrollX = (uint8_t)c->x;

  // Have we scrolled past a tile boundary?
  if ((prevX & ~(TILE_WIDTH - 1)) != (Screen.scrollX & ~(TILE_WIDTH - 1))) {
    int16_t cxt = c->x / TILE_WIDTH;
    if ((uint8_t)(Screen.scrollX - prevX) < (uint8_t)(prevX - Screen.scrollX))
      Level_drawColumn(c->level, cxt + 29, cxt + 29);
    else
      Level_drawColumn(c->level, cxt + 30, cxt - 2);
  }
}
```

Since this function is called every frame, before we scroll the screen to the current camera position (in pixels) by setting Screen.scrollX, we store its previous value. Then we check to see if we just scrolled past a tile boundary, which occurs every 8 pixels, by masking off the lowest three bits (forcing them to zero) from both prevX and Screen.scrollX and then comparing only their five highest bits with each other. If those high bits are the same, we haven't moved off the current tile yet, so we don't need to call Level_drawColumn, but if they are different then we know we just scrolled past a tile boundary and we need to refresh the data in a hidden column.[2] Be aware that this code assumes you aren't going to move the camera by more than 8 pixels between calls to Camera_update, but that's a fair assumption for most games.

If we crossed a tile boundary, we convert the current camera's x position from pixels into tiles, calculate which direction we just scrolled, and call Level_drawColumn on the column farthest away from the viewport in the direction we scrolled (column 29 or 30 in the conceptual diagrams) passing the appropriate value for realX.

And that's everything you need to get bi-directional horizontal mode 3 scrolling working on the Uzebox—in under 100 lines of actual code!

Go to your project's default directory, type make and you should have a horiz.uze file that you can run in either the uzem emulator, or the cuzebox emulator, or you can copy it to an SD card and run it on your real hardware. If you need help doing any of this, you can watch my video tutorial series.

[2]Thanks to CunningFellow for this suggestion

A complete version of this project, including tile resources, and a `Makefile` is available online at:

https://github.com/artcfox/uzebox-mode-3-with-scrolling-guide

In order for the included `Makefile` to automatically find the Uzebox kernel and build tools, you should run:

```
git clone https://github.com/artcfox/uzebox-mode-3-with-scrolling-guide
```

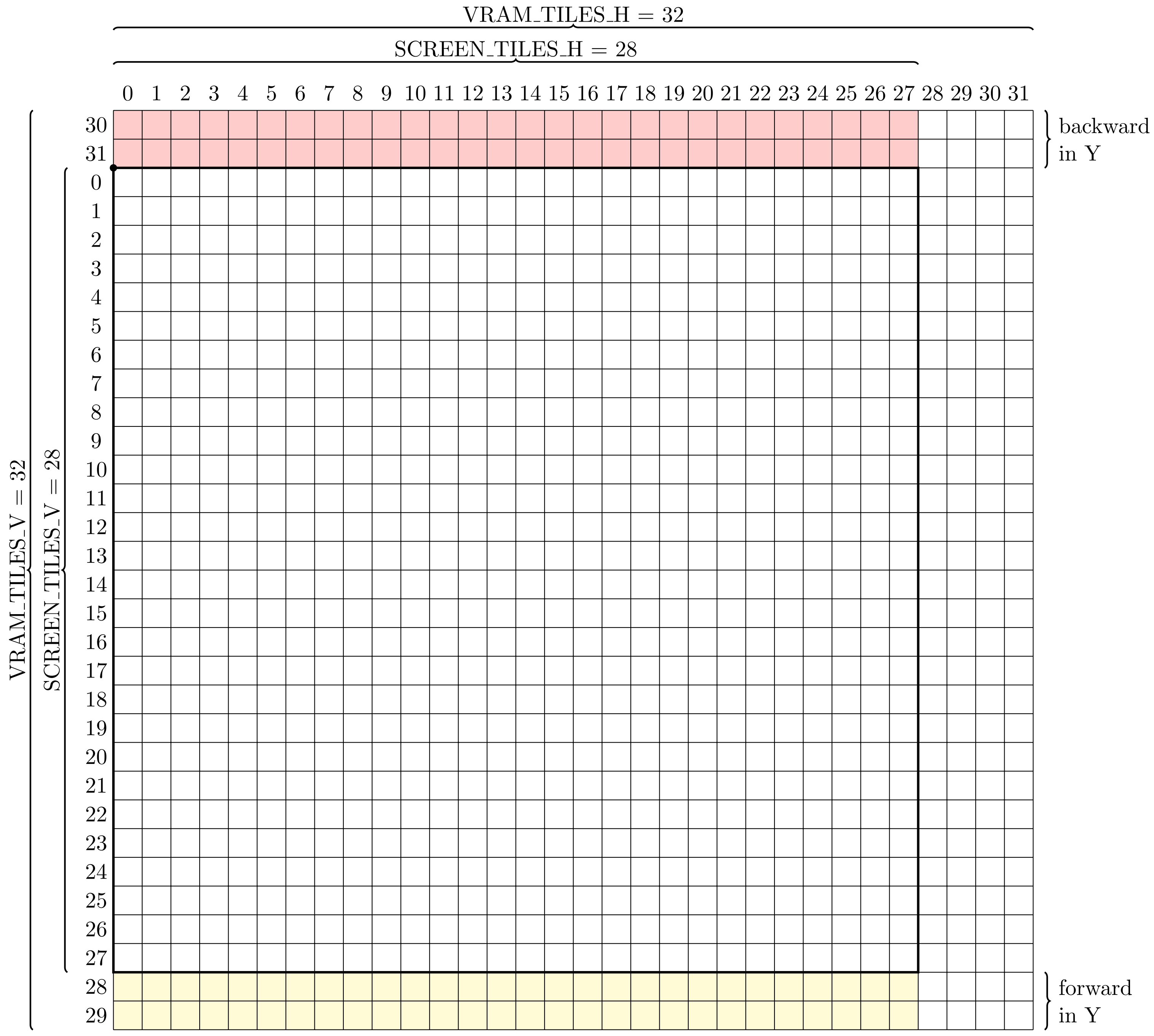
from within your main uzebox directory, and not within an additional subdirectory, since this repository contains multiple projects that expect to find the kernel and build tools based on a relative path a specific number of directory levels away.

Vertical Scrolling

Conceptual Diagrams

In the case of vertical only scrolling, our diagram will consist of a flattened VRAM torus rotated so the viewport is only centered vertically, because this makes it easier to conceptualize which rows are forward rows and which are backward rows. The dot represents the origin of VRAM and the origin of the screen.

Since we are only scrolling vertically, we can skip writing tile data to columns 28, 29, 30, and 31.

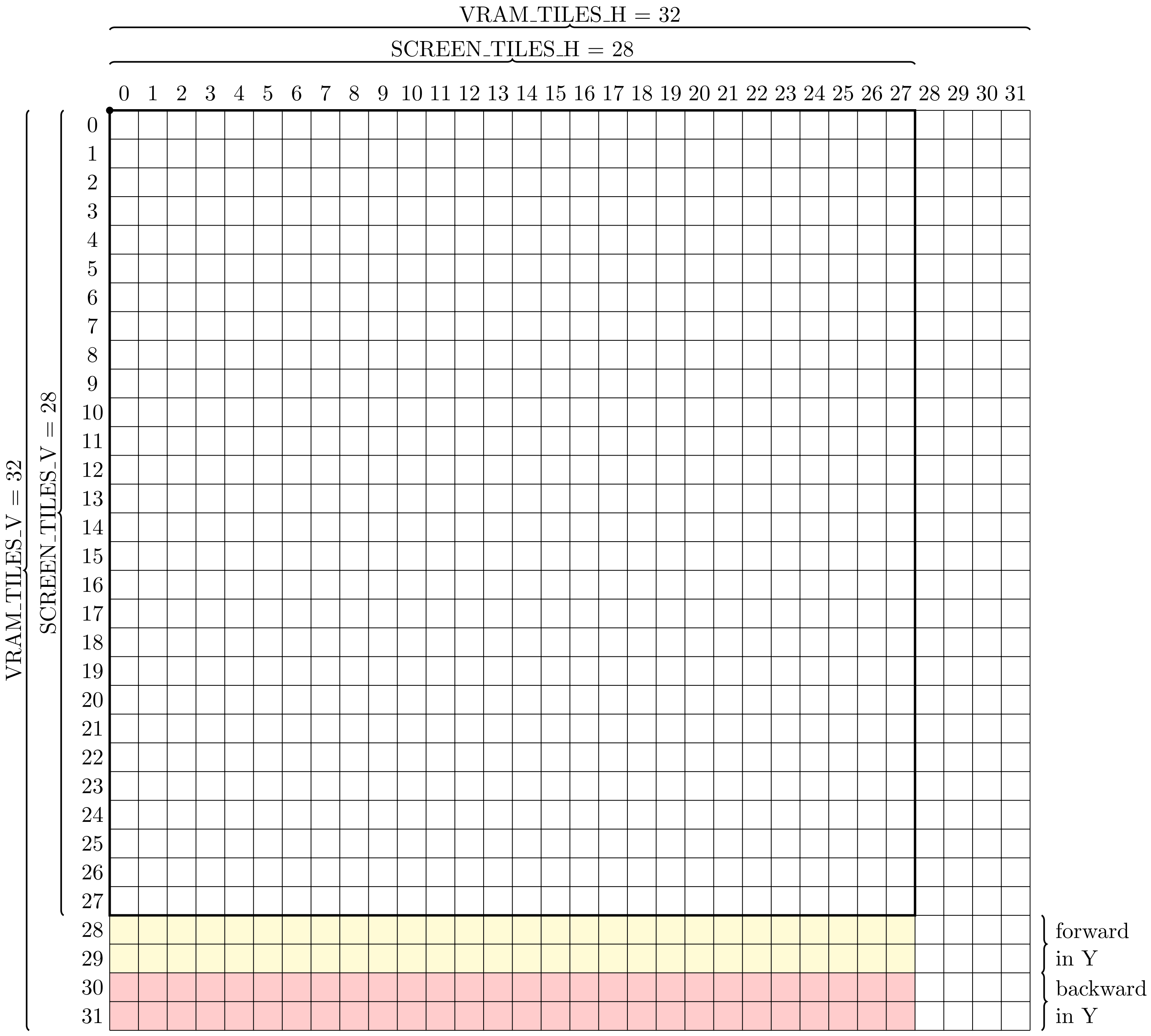


🟥 hidden level tiles backward in the Y direction

🟨 hidden level tiles forward in the Y direction

To help us populate the contents of VRAM programatically, it's useful to view the diagram above with the VRAM array wrapped so the origin appears in the top left corner, rather than wrapped to make everything symmetrical. Both views are equivalent, but this view should make it easier to see which tiles need to be populated using offsets and modulo arithmetic.

Keep in mind that when you are writing new tiles from your level data into hidden VRAM upon scrolling, you only need to write to the parts of hidden VRAM that are the farthest away from your viewport, which in this case would be rows 29 and 30. The other hidden rows will already be populated with the correct tiles.



🟥 hidden level tiles backward in the Y direction

🟨 hidden level tiles forward in the Y direction

Code

Vertical scrolling is almost identical to horizontal scrolling, except in a few places. If you are reading this section after reading the previous section on horizontal scrolling it will probably feel boring and repetitive. If so, feel free to skip it and try implementing the code yourself based on the horizontal version. I want to keep the same level of detail here as before, so anyone looking to just implement vertical scrolling can find all the information they need in one place.

This code requires[3] your `Makefile` to have the following `KERNEL_OPTIONS` defined:

```
-DVIDEO_MODE=3 -DSCROLLING=1 -DSCREEN_TILES_V=28 -DSCREEN_TILES_H=28 -DVRAM_TILES_V=32
```

Very specific linker options are required for mode 3 with scrolling to function, so your `Makefile` must also include:

```
## Adjust the .data value to be 0x800100+VRAM_TILES_H*VRAM_TILES_V
LDFLAGS += -Wl,--section-start,.noinit=0x800100 -Wl,--section-start,.data=0x800500
```

I'm assuming you already have a Uzebox development environment installed and configured, and that you know how to create a Uzebox-specific tileset and define map variables in your `tileset.xml` file. If not, I created a video tutorial series that can walk you through this entire process.

The code in this section assumes your `tileset.xml` file looks like this:

```
<?xml version="1.0" ?>
<gfx-xform version="1">
  <input file="../data/tileset.png" type="png" tile-width="8" tile-height="8" />
  <output file="../data/tileset.inc" remove-duplicate-tiles="true">
    <tiles var-name="tileset"/>
    <maps pointers-size="8">
      <map var-name="map_vert_level" left="0" top="0" width="28" height="56" />
    </maps>
  </output>
</gfx-xform>
```

Your map can have a different height, but its width should be 28 tiles, and the code assumes your map variable is called `map_vert_level`. You are free to define additional levels, just give your new map variables unique names, and modify your code accordingly.

We will start with the list of `#include` files and the contents of the `main()` function, and then we can go back and fill in the rest. This should also give you a good overview of how the code will be structured.

Create a file called `vert.c` using your favorite text editor, and write the following:

```
int main()
{
  SetTileTable(tileset);

  LEVEL l;
  CAMERA c;

  ClearVram();
  Level_initFromMap(&l, map_vert_level);
  Camera_init(&c, &l);
  Camera_fillVram(&c);
```

[3]While `-DSCREEN_TILES_V=28` and `-DSCREEN_TILES_H=28` aren't strict requirements, they are the default values for mode 3 with scrolling, so for tutorial purposes that's what the code will assume. My philosophy is to start with the basics, get that working 100%, then refine, refactor, and make improvements. Once the initial code is written and proven to be correct, it'll be easier to spot the patterns, and then we can go back and make the code generic enough to handle different screen sizes. That is also the reason why this tutorial is not using any form of level compression. Solve one thing at a time, and you'll be able to isolate any bugs quicker.

```
  for (;;) {
    uint16_t held = ReadJoypad(0);
    if (held & BTN_UP)
      Camera_moveTo(&c, c.y - 1);
    else if (held & BTN_DOWN)
      Camera_moveTo(&c, c.y + 1);

    Camera_update(&c);

    WaitVsync(1);
  }
}
```

As you can see, we have a `LEVEL` object and a `CAMERA` object, and each one has various functions associated with them. This allows us to keep everything organized, and it makes the code self-descriptive. Go ahead and read it line by line; you should get a pretty good idea of exactly what will happen.

The `LEVEL` object

Next we will define our `LEVEL` object. We want it to automatically keep track of the level's width, height, level data, and an offset into the level data. In this tutorial we will use the offset to skip the width and height that `gconvert` automatically adds to the beginning of our map data, but this offset can also be useful in a scenario where you have the data for multiple levels all packed together in a single array.

Below the `#include` lines, define the following struct:

```
typedef struct {
  uint16_t width;
  uint16_t height;
  uint16_t offset;
  const char *data;
} __attribute__ ((packed)) LEVEL;
```

The `__attribute__ ((packed))` is not strictly required, but it tells the compiler to not leave any padding between members of the struct.

Now define the `Level_initFromMap` function:

```
void Level_initFromMap(LEVEL *l, const char *map)
{
  l->width = pgm_read_byte(&map[0]);
  l->height = pgm_read_byte(&map[1]);
  l->offset = 2;
  l->data = map;
}
```

The first parameter is a pointer to the `LEVEL` object, and the second is a pointer to the level data. In our case, this will be the name of the map variable we defined in our `tileset.xml` file. Since that array data is actually stored inside the flash memory (PROGMEM) of the AVR rather than RAM, we need to use the `pgm_read_byte` function to access it.

In the future if you want to store your level data differently, i.e., not in a `gconvert` produced map, you can still use the `LEVEL` object, all you will need to do is add a different `Level_init` function that populates the `LEVEL` object appropriately.

Next, define a function that allows us to get a tile from the level based on an index into the data:

```
static inline uint8_t Level_getTileAt(LEVEL *l, uint16_t index)
{
  return pgm_read_byte(&l->data[l->offset + index]);
}
```

Make sure you define the function as `static inline` since it will be called from inside tight loops, and we don't want to incur the overhead of a function call every time around the loop. Again, we're using `pgm_read_byte` to access the level data, and we're adding the level offset we previously stored in the `LEVEL` object to the index given to us.

The last thing we need to complete our `LEVEL` object is to add the `Level_drawRow` function:

```
void Level_drawRow(LEVEL *l, uint8_t y, int16_t realY)
{
  if (realY < 0 || (uint16_t)realY > l->height - 1)
    return;

  uint8_t ty = y % VRAM_TILES_V;

  for (uint8_t i = 0; i < SCREEN_TILES_H; ++i) {
    uint16_t index = realY * l->width + i;
    SetTile(i, ty, Level_getTileAt(l, index));
  }
}
```

Here is where things start to get interesting, because we need to give it two y values. If you refer back to the conceptual diagrams, you'll see that `y` is needed to select which row of VRAM to populate, and `realY` is needed to select which row to read from in our level array in order to get the proper tile values. The variable `y` needs to wrap around the torus, but `realY` cannot because it is a reference to a specific row of level tiles.

This is also why `realY` is a signed data type, since it's possible for it to go negative when we are at the beginning edge of the level and trying to look backward.

The first line in the function ensures we don't read before the beginning of the level array when we are scrolled to the beginning of the level and try to look backward, and that we don't read past the end of the level array when we are scrolled to the end of the level and try to look forward.

To properly index into the VRAM array as if it were a torus, we need to use `y % VRAM_TILES_V` as the second parameter to `SetTile`, but we store it in the `ty` variable outside the loop so this calculation only has to happen once. This allows us to pass any `y` value we want into the `Level_drawRow` function, and it will automatically be wrapped properly onto a valid VRAM row. Since this is the only place in the fuction we use `y`, we can get away with making its type `uint8_t`, because forcing 8-bit truncation won't change the result of the modulo by `VRAM_TILES_V`.

Next we loop over every visible column in the screen calculating an index into our level array, based on `realY` (which may be looking backward or forward) and call `SetTile` with the tile number returned from the `Level_getTileAt` function.

Notice that since we abstracted the concept of retrieving a tile from the level by calling the `Level_getTileAt` function, no changes to the `Level_drawRow` function are required if, in the future, you decide to compress your level data. You still may want to make changes for performance reasons, but don't do that until you've fully debugged your level compression scheme!

The `CAMERA` object

Now that the `LEVEL` object is complete, we can define our `CAMERA` object. This one is so simple that you might wonder why we are even creating an object for it, but you'll appreciate the encapsulation later as you extend it and add features. Since we are only going to be scrolling vertically, it just needs to keep track of the camera's y position in pixels across the height of the level, and store a pointer to the `LEVEL` object, so internally it can query the level and call its various functions.

Just below the `Level_drawRow` function, define the following struct:

```
typedef struct {
  int16_t y;
  LEVEL *level;
} __attribute__ ((packed)) CAMERA;
```

Next, define the `Camera_init` function:

```
void Camera_init(CAMERA *c, LEVEL *l)
{
  c->level = l;
  c->y = 0;
  Screen.scrollX = 0;
  Screen.scrollY = (uint8_t)c->y;
}
```

Its implementation is self-explanatory; we store the level pointer, set the initial y position of the camera to zero, set the screen's y scrolling position to the camera's y position (zero), and set the screen's x scrolling position to zero to ensure it is in a known state.

As your game gets more developed you can imagine designing levels where your camera doesn't start at the far top of the level. A clean way to implement that would be to find a good place to store the initial y position of the camera for each level, have your customized `Level_init` function read it for that level (extend your `LEVEL` struct to store it), and finally query that value from your `LEVEL` object from within the `Camera_init` function, replacing the blind assignment to zero.

Now define a function we can use to set the camera's position:

```
void Camera_moveTo(CAMERA *c, int16_t y)
{
  if (y < 0) {
    c->y = 0;
  } else {
    int16_t yMax = (c->level->height - SCREEN_TILES_V) * TILE_HEIGHT;
    if (y > yMax)
      c->y = yMax;
    else
      c->y = y;
  }
}
```

This function will automatically clamp any y value we call it with to a valid camera position for the currently loaded level. Remember that the camera's y position represents pixels down the level. For the assignment to `yMax`, the level height is stored in tiles, so we subtract the screen height (also in tiles) and then multiply the result by `TILE_HEIGHT` to convert it into pixels. Because the origin of the viewport is at top left of the screen, the camera can't get any closer than a screen's height away from the bottom edge of the level.

Add the `Camera_fillVram` function next:

```
void Camera_fillVram(CAMERA *c)
{
  int16_t cyt = c->y / TILE_HEIGHT;

  for (uint8_t i = 0; i < VRAM_TILES_V - 2; ++i)
    Level_drawRow(c->level, cyt + i, cyt + i);

  Level_drawRow(c->level, cyt + 30, cyt - 2);
  Level_drawRow(c->level, cyt + 31, cyt - 1);
}
```

The first thing it does is convert the current camera's y position from pixels into tiles, and stores it in the variable `cyt`.

Next, refer to the second conceptual diagram to see which rows we can draw "normally," meaning by passing the same `y` and `realY` value to the `Level_drawRow` function (hint: rows 0 through 29). Even though rows 28 and 29 are inside the hidden portion of VRAM, we can still index their VRAM row and the relative position of the tiles in the level array by adding one for each row beyond the camera's current position (represented by the dot in the diagram) that we move. That's why the `for` loop can go from `0` to `VRAM_TILES_V - 2`.

For the remaining two rows, we still need to increment their indices into VRAM (`cyt + 30` and `cyt + 31`), but at the same time we need to reference rows from the level array that are backward in y, relative to the camera's current position, hence the `cyt - 2` and `cyt - 1`.

If you are confused as to how much each of those last two rows needs to look backward in the level array, refer back to the first conceptual diagram where the viewport is centered, and match the row numbers up when moving to the second conceptual diagram. Now you understand why I drew both diagrams.

Also notice that we didn't have to deal with any VRAM wrapping, or worry about referencing positions outside the bounds of the level array here, because that's already been handled for us inside the `Level_drawRow` function.

The only function we haven't defined yet is the `Camera_update` function which will scroll the screen, and draw the necessary row into the hidden portion of VRAM for us, so define that now:

```
void Camera_update(CAMERA *c)
{
  uint8_t prevY = Screen.scrollY;
  Screen.scrollY = (uint8_t)c->y;

  // Have we scrolled past a tile boundary?
  if ((prevY & ~(TILE_HEIGHT - 1)) != (Screen.scrollY & ~(TILE_HEIGHT - 1))) {
    int16_t cyt = c->y / TILE_HEIGHT;
    if ((uint8_t)(Screen.scrollY - prevY) < (uint8_t)(prevY - Screen.scrollY))
      Level_drawRow(c->level, cyt + 29, cyt + 29);
    else
      Level_drawRow(c->level, cyt + 30, cyt - 2);
  }
}
```

Since this function is called every frame, before we scroll the screen to the current camera position (in pixels) by setting `Screen.scrollY`, we store its previous value. Then we check to see if we just scrolled past a tile boundary, which occurs every 8 pixels, by masking off the lowest three bits (forcing them to zero) from both `prevY` and `Screen.scrollY` and then comparing only their five highest bits with each other. If those high bits are the same, we haven't moved off the current tile yet, so we don't need to call `Level_drawRow`, but if they are different then we know we just scrolled past a tile boundary and we need to refresh the data in a hidden row.[4] Be aware that this code assumes you aren't going to move the camera by more than 8 pixels between calls to `Camera_update`, but that's a fair assumption for most games.

If we crossed a tile boundary, we convert the current camera's y position from pixels into tiles, calculate which direction we just scrolled, and call `Level_drawRow` on the row farthest away from the viewport in the direction we scrolled (row 29 or 30 in the conceptual diagrams) passing the appropriate value for `realY`.

And that's everything you need to get bi-directional vertical mode 3 scrolling working on the Uzebox—in under 100 lines of actual code!

Go to your project's `default` directory, type `make` and you should have a `vert.uze` file that you can run in either the `uzem` emulator, or the `cuzebox` emulator, or you can copy it to an SD card and run it on your real hardware. If you need help doing any of this, you can watch my video tutorial series.

A complete version of this project, including tile resources, and a `Makefile` is available online at:

https://github.com/artcfox/uzebox-mode-3-with-scrolling-guide

[4]Thanks to CunningFellow for this suggestion

In order for the included `Makefile` to automatically find the Uzebox kernel and build tools, you should run:

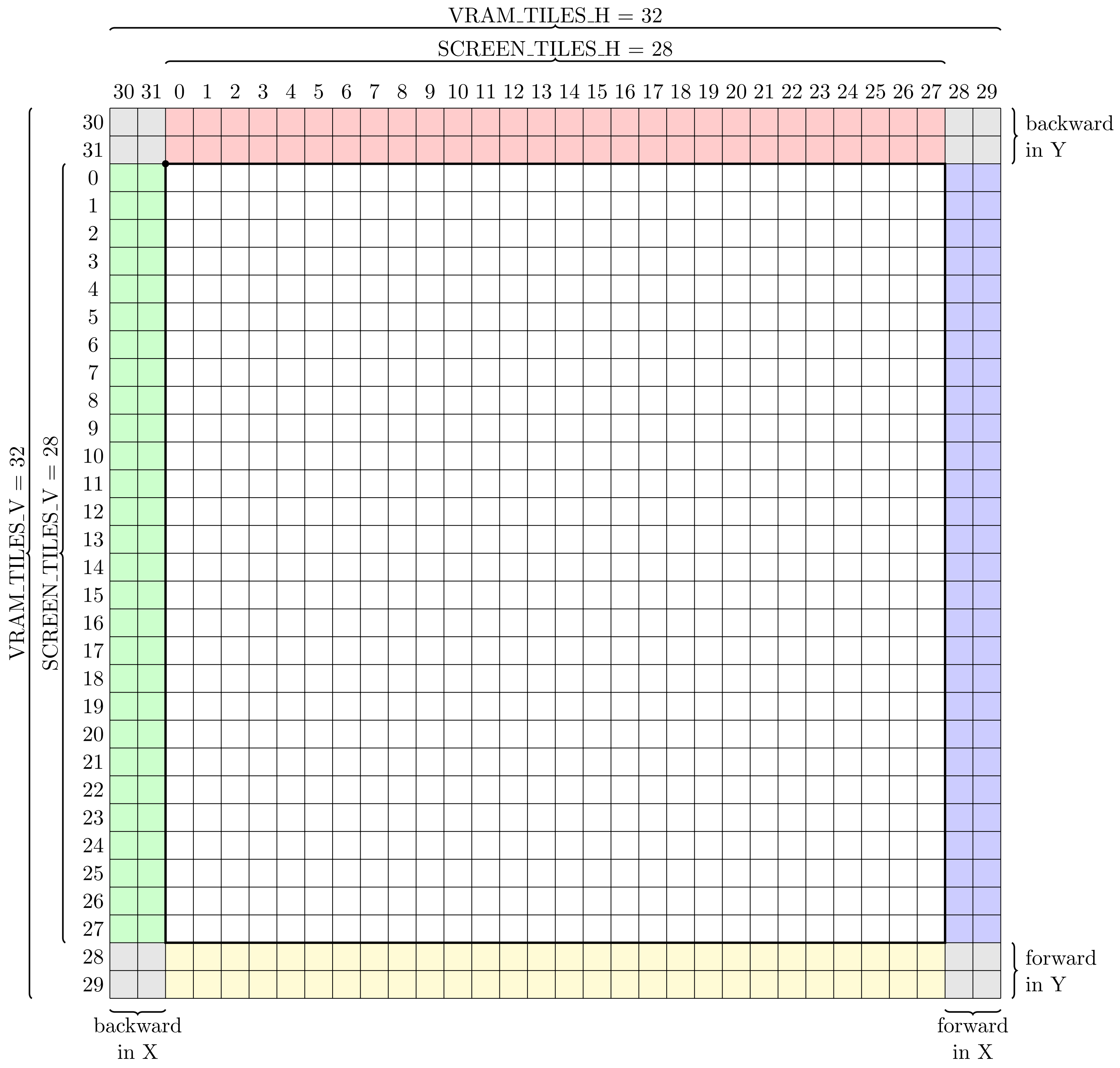
```
git clone https://github.com/artcfox/uzebox-mode-3-with-scrolling-guide
```

from within your main uzebox directory, and not within an additional subdirectory, since this repository contains multiple projects that expect to find the kernel and build tools based on a relative path a specific number of directory levels away.

Horizontal and Vertical Scrolling

Conceptual Diagrams

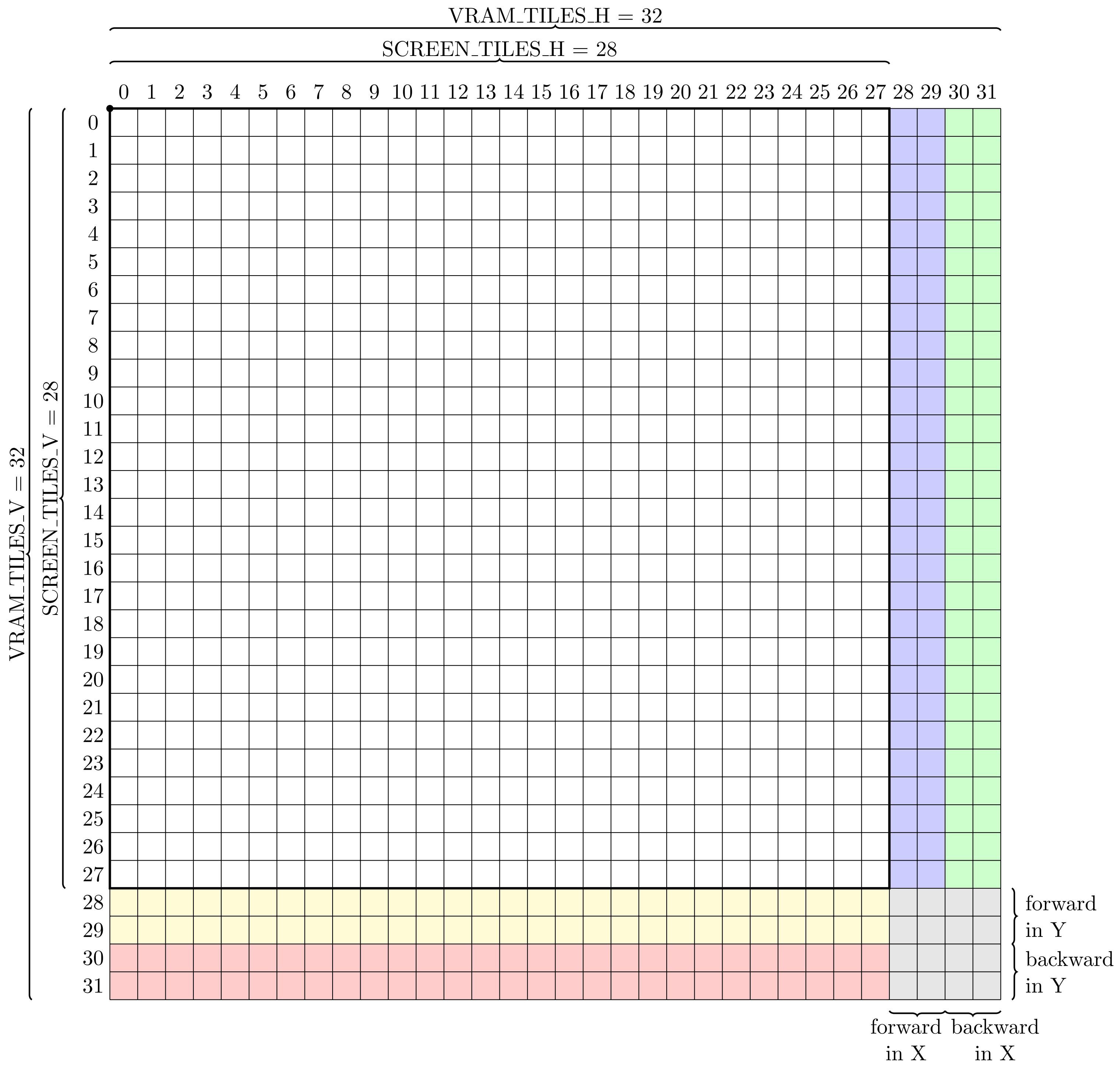
This is the granddaddy of all scrolling modes! Ensure you have a good understanding of how the previous scrolling modes work, because this one is the most complicated, and can be difficult to implement correctly.



🟩 hidden level tiles backward in the X direction
🟥 hidden level tiles backward in the Y direction
🟦 hidden level tiles forward in the X direction
🟨 hidden level tiles forward in the Y direction
⬜ various combinations of hidden level tiles forward and backward in the X and Y directions

To populate the contents of VRAM programatically, it's useful to view the same diagram as above with the VRAM array wrapped differently so the origin appears in the top left corner, rather than wrapped to make everything symmetrical. Both views are equivalent, but this view should make the offsets and modulo arithmetic you'll need to use more apparent.

Keep in mind when you are writing tiles from your level array into VRAM, you only need to write to the rows and columns in hidden VRAM that are the farthest away from your viewport, which in this case would be rows 29 and 30, and columns 29 and 30 (depending on the direction you are scrolling). The other hidden rows and columns will already be populated with the correct tile data.



🟩	hidden level tiles backward in the X direction	🟥	hidden level tiles backward in the Y direction
🟦	hidden level tiles forward in the X direction	🟨	hidden level tiles forward in the Y direction
⬜	various combinations of hidden level tiles forward and backward in the X and Y directions		

Code

For scrolling both horizontally and vertically at the same time, one might think it's possible to just add both the `Level_drawColumn` function from the horizontal scrolling example, and the `Level_drawRow` function from the vertical scrolling example, and then call each function when appropriate, but that will not work. If you try to force the issue, you'll end up with something that mostly works, but it will have corruption (tiles that are missing, or tiles that appear where they shouldn't) as you try to scroll around both axes at the same time. You might notice that the corruption changes depending on whether you draw your rows or columns first. You might also try to "patch over" that corruption by writing to visible areas of VRAM (specifically column 0 and/or row 0), but that's not the solution. The corruption occurs because trying to reuse those functions causes the wrong level tiles to be written into some of the gray squares in the above conceptual diagrams, and when you manage to scroll one of those squares onto the screen, you won't see what you were expecting.[5]

If you are reading this section after reading the previous sections on horizontal and/or vertical scrolling, I apologize if parts of it sound repetitive, but I want to keep the same level of detail here as before, so anyone who jumps right to this section can find all the information they need in one place.

This code requires[6] your `Makefile` to have the following `KERNEL_OPTIONS` defined:

```
-DVIDEO_MODE=3 -DSCROLLING=1 -DSCREEN_TILES_V=28 -DSCREEN_TILES_H=28 -DVRAM_TILES_V=32
```

Very specific linker options are required for mode 3 with scrolling to function, so your `Makefile` must also include:

```
## Adjust the .data value to be 0x800100+VRAM_TILES_H*VRAM_TILES_V
LDFLAGS += -Wl,--section-start,.noinit=0x800100 -Wl,--section-start,.data=0x800500
```

I'm assuming you already have a Uzebox development environment installed and configured, and that you know how to create a Uzebox-specific tileset and define map variables in your `tileset.xml` file. If not, I created a video tutorial series that can walk you through this entire process.

The code in this section assumes your `tileset.xml` file looks like this:

```
<?xml version="1.0" ?>
<gfx-xform version="1">
  <input file="../data/tileset.png" type="png" tile-width="8" tile-height="8" />
  <output file="../data/tileset.inc" remove-duplicate-tiles="true">
    <tiles var-name="tileset"/>
    <maps pointers-size="8">
      <map var-name="map_horiz_vert_level" left="0" top="0" width="60" height="56" />
```

[5]Can you guess how I know about this corruption in such great detail? Debugging it involved me using the whisper ports to write character and hex values to the console of `uzem`. First I had `uzem` record my inputs so I could run the exact same test more than once. Then I had `uzem` log the frame number along with every call to `SetTile`, the `x`, the `y`, and the `index` into my level array, indicating whether that call came from `Level_drawRow` or `Level_drawColumn`. Then I ended up porting my game code to Linux, simulating the VRAM of the Uzebox and the calls to `SetTile` so I could get a picture of what the entire contents of VRAM (including the hidden areas) looked like at any given time. Then I modified the Uzebox version to instead output the camera's x and y position to the console as a C-style array, so the camera in my Linux port could track along the exact same path and reveal the same screen corruption. Once I saw the corruption happen in the Linux version, I looked back frame by frame until I found the first frame that got corrupted, and then I cross-referenced that frame number with the log data of every call to `SetTile` I had generated previously using the Uzebox version. At this point I knew which `x` and `y` coordinate was getting corrupted on which frame, and when I looked at the logs for `SetTile` for that coordinate, for that frame, I found that the VRAM location was being written to by both `Level_drawRow` and `Level_drawColumn`, but each of them used a different index into the level array, and those indices were 32 apart from each other. I have to say it was nice being able to step through my code using a visual debugger and to have a full arsenal of other profiling/debugging tools at my disposal, but that was not an experience I want to repeat anytime soon. And that's the story behind why this guide has those conceptual diagrams drawn out—because I needed them in order to get my implementation working correctly!

[6]While `-DSCREEN_TILES_V=28` and `-DSCREEN_TILES_H=28` aren't strict requirements, they are the default values for mode 3 with scrolling, so for tutorial purposes that's what the code will assume. My philosophy is to start with the basics, get that working 100%, then refine, refactor, and make improvements. Once the initial code is written and proven to be correct, it'll be easier to spot the patterns, and then we can go back and make the code generic enough to handle different screen sizes. That is also the reason why this tutorial is not using any form of level compression. Solve one thing at a time, and you'll be able to isolate any bugs quicker.

```
    </maps>
  </output>
</gfx-xform>
```

In this case, your map can have both a different width and height, but the code in this section assumes your map variable is called `map_horiz_vert_level`. You are free to define additional levels, just give your new map variables unique names, and modify your code accordingly.

We will start with the list of `#include` files and the contents of the `main()` function, and then we can go back and fill in the rest. This should also give you a good overview of how the code will be structured.

Create a file called `horiz-vert.c` using your favorite text editor, and write the following:

```
int main()
{
  SetTileTable(tileset);

  LEVEL l;
  CAMERA c;

  ClearVram();
  Level_initFromMap(&l, map_horiz_vert_level);
  Camera_init(&c, &l);
  Camera_fillVram(&c);

  for (;;) {
    uint16_t held = ReadJoypad(0);
    if (held & BTN_LEFT)
      Camera_moveTo(&c, c.x - 1, c.y);
    else if (held & BTN_RIGHT)
      Camera_moveTo(&c, c.x + 1, c.y);

    if (held & BTN_UP)
      Camera_moveTo(&c, c.x, c.y - 1);
    else if (held & BTN_DOWN)
      Camera_moveTo(&c, c.x, c.y + 1);

    Camera_update(&c);

    WaitVsync(1);
  }
}
```

As you can see, we have a `LEVEL` object and a `CAMERA` object, and each one has various functions associated with them. This allows us to keep everything organized, and it makes the code self-descriptive. Go ahead and read it line by line; you should get a pretty good idea of exactly what will happen.

The `LEVEL` object

Next we will define our `LEVEL` object. We want it to automatically keep track of the level's width, height, level data, and an offset into the level data. In this tutorial we will use the offset to skip the width and height that `gconvert` automatically adds to the beginning of our map data, but this offset can also be useful in a scenario where you have the data for multiple levels all packed together in a single array.

Below the `#include` lines, define the following struct:

```
typedef struct {
  uint16_t width;
  uint16_t height;
  uint16_t offset;
  const char *data;
} __attribute__ ((packed)) LEVEL;
```

The `__attribute__ ((packed))` is not strictly required, but it tells the compiler to not leave any padding between members of the struct.

Now define the `Level_initFromMap` function:

```
void Level_initFromMap(LEVEL *l, const char *map)
{
  l->width = pgm_read_byte(&map[0]);
  l->height = pgm_read_byte(&map[1]);
  l->offset = 2;
  l->data = map;
}
```

The first parameter is a pointer to the `LEVEL` object, and the second is a pointer to the level data. In our case, this will be the name of the map variable we defined in our `tileset.xml` file. Since that array data is actually stored inside the flash memory (PROGMEM) of the AVR rather than RAM, we need to use the `pgm_read_byte` function to access it.

In the future if you want to store your level data differently, i.e., not in a `gconvert` produced map, you can still use the `LEVEL` object, all you will need to do is add a different `Level_init` function that populates the `LEVEL` object appropriately.

Next, define a function that allows us to get a tile from the level based on an index into the data:

```
static inline uint8_t Level_getTileAt(LEVEL *l, uint16_t index)
{
  return pgm_read_byte(&l->data[l->offset + index]);
}
```

Make sure you define the function as `static inline` since it will be called from inside tight loops, and we don't want to incur the overhead of a function call every time around the loop. Again, we're using `pgm_read_byte` to access the level data, and we're adding the level offset we previously stored in the `LEVEL` object to the index given to us.

Now this is where things start to diverge from the other scrolling modes, and we'll need to refer to the conceptual diagrams to ensure we get everything correct. The following two functions are going to be the most complicated functions in this entire guide.

We'll start by defining the `Level_drawColumn` function:

```
void Level_drawColumn(LEVEL *l, uint8_t x, uint16_t y, int16_t realX)
{
  if (realX < 0 || (uint16_t)realX > l->width - 1)
    return;

  uint8_t tx = x % VRAM_TILES_H;

  for (uint8_t i = 0; i < VRAM_TILES_V - 2; ++i) {
    if ((y + i) > l->height - 1)
      break;
    uint16_t index = (y + i) * l->width + realX;
    SetTile(tx, (y + i) % VRAM_TILES_V, Level_getTileAt(l, index));
  }
  if (y > 1)
    SetTile(tx, (y + 30) % VRAM_TILES_V,
        Level_getTileAt(l, ((int16_t)y - 2) * l->width + realX));
  if (y > 0)
    SetTile(tx, (y + 31) % VRAM_TILES_V,
        Level_getTileAt(l, ((int16_t)y - 1) * l->width + realX));
}
```

The parameter `x` only needs to be used as an offset when indexing into the wrapped VRAM array, so its data type can be `uint8_t`. The parameter `y` does double-duty, in that it is used as an offset when indexing the wrapped

VRAM array in the calls to `SetTile`, and it is also used as an offset when building the index into the level array. The final parameter, `realX` is only used to help build the index into the level array, so it never gets wrapped, and it must have a signed data type because it may be backward looking, and thus go negative when looking backward while already at the beginning of the level array (which is detected and prevented).

First we perform the bounds check on `realX`, and wrap `x` outside the `for` loop. Then we loop over all the visible rows plus two hidden rows (in yellow on the diagram), perform a bounds check on the `y` offset added to the index variable `i` (exiting the loop before we run off the end of the level array), and then compute an index into the level array based off `realX` and using the full value of `y` as an offset, plus our index variable, `i`. Then we call `SetTile`, with the level data stored at that index.

Finally, we individually call `SetTile` for the last two tiles in the column (in red on the diagram), performing a bounds check on `y`, before indexing the level array backward in y, by one or two tiles.

Next, we need to define the `Level_drawRow` function. It's the perfect analogue to the `Level_drawColumn` function, except it does everything on the opposite axis. It can really help to study the functions side-by-side, and spot the differences.

```
void Level_drawRow(LEVEL *l, uint16_t x, uint8_t y, int16_t realY)
{
  if (realY < 0 || (uint16_t)realY > l->height - 1)
    return;

  uint8_t ty = y % VRAM_TILES_V;

  for (uint8_t i = 0; i < VRAM_TILES_H - 2; ++i) {
    if ((x + i) > l->width - 1)
      break;
    uint16_t index = realY * l->width + (x + i);
    SetTile((x + i) % VRAM_TILES_H, ty, Level_getTileAt(l, index));
  }
  if (x > 1)
    SetTile((x + 30) % VRAM_TILES_H, ty,
        Level_getTileAt(l, realY * l->width + ((int16_t)x - 2)));
  if (x > 0)
    SetTile((x + 31) % VRAM_TILES_H, ty,
        Level_getTileAt(l, realY * l->width + ((int16_t)x - 1)));
}
```

The parameter `x` does double-duty, in that it is used as an offset when indexing the wrapped VRAM array in the calls to `SetTile`, and it is also used as an offset when building the index into the level array. The parameter `y` only needs to be used as an offset when indexing into the wrapped VRAM array, so its data type can be `uint8_t`. The final parameter, `realY` is only used to help build the index into the level array, so it never gets wrapped, and it must have a signed data type because it may be backward looking, and thus go negative when looking backward while already at the beginning of the level array (which is detected and prevented).

First we perform the bounds check on `realY`, and wrap `y` outside the `for` loop. Then we loop over all the visible columns plus two hidden columns (in blue on the diagram), perform a bounds check on the `x` offset added to the index variable `i` (exiting the loop before we run off the end of the level array), and then compute an index into the level array based off `realY` and using the full value of `x` as an offset, plus our index variable, `i`. Then we call `SetTile`, with the level data stored at that index.

Finally, we individually call `SetTile` for the last two tiles in the row (in green on the diagram), performing a bounds check on `x`, before indexing the level array backward in x, by one or two tiles.

The `CAMERA` object

Now that the `LEVEL` object is complete, we can define our `CAMERA` object. Since we are going to be scrolling horizontally and vertically, it needs to keep track of the camera's x position in pixels across the width of the level, the y position in pixels across the height of the level, and store a pointer to the `LEVEL` object, so internally it can query the level and call its various functions.

Just below the `Level_drawRow` function, define the following struct:

```
typedef struct {
  int16_t x;
  int16_t y;
  LEVEL *level;
} __attribute__ ((packed)) CAMERA;
```

Next, define the `Camera_init` function:

```
void Camera_init(CAMERA *c, LEVEL *l)
{
  c->level = l;
  c->x = c->y = 0;
  Screen.scrollX = (uint8_t)c->x;
  Screen.scrollY = (uint8_t)c->y;
}
```

The implementation is self-explanatory; we store the level pointer, set the initial x and y position of the camera to zero, and set the screen's x and y scrolling position to the camera's x and y positions (zero).

As your game gets more developed you can imagine designing levels where your camera doesn't start at the top left of the level. A clean way to implement that would be to find a good place to store the initial x and y coordinates of the camera for each level, have your customized `Level_init` function read them for that level (extend your `LEVEL` struct to store them), and finally query the x and y coordinates from your `LEVEL` object from within the `Camera_init` function, replacing the blind assignments to zero.

Now define a function we can use to set the camera's position:

```
void Camera_moveTo(CAMERA *c, int16_t x, int16_t y)
{
  if (x < 0) {
    c->x = 0;
  } else {
    int16_t xMax = (c->level->width - SCREEN_TILES_H) * TILE_WIDTH;
    if (x > xMax)
      c->x = xMax;
    else
      c->x = x;
  }

  if (y < 0) {
    c->y = 0;
  } else {
    int16_t yMax = (c->level->height - SCREEN_TILES_V) * TILE_HEIGHT;
    if (y > yMax)
      c->y = yMax;
    else
      c->y = y;
  }
}
```

This function will automatically clamp any x and y value we call it with to a valid camera position for the currently loaded level. Remember that the camera's x position represents pixels across the level, and the y position represents pixels down the level. For the assignment to `xMax`, the level width is stored in tiles, so we subtract the screen width (also in tiles) and then multiply the result by `TILE_WIDTH` to convert it into pixels. A similar computation is made for `yMax`, just using height instead of width.

Because the origin of the viewport is at top left of the screen, the camera can't get any closer than a screen's width away from the right edge of the level, and no more than a screen's height away from the bottom edge of the level.

Add the `Camera_fillVram` function next:

```
void Camera_fillVram(CAMERA *c)
{
  int16_t cxt = c->x / TILE_WIDTH;
  int16_t cyt = c->y / TILE_HEIGHT;

  for (uint8_t i = 0; i < VRAM_TILES_H - 2; ++i)
    Level_drawColumn(c->level, cxt + i, cyt, cxt + i);

  Level_drawColumn(c->level, cxt + 30, cyt, cxt - 2);
  Level_drawColumn(c->level, cxt + 31, cyt, cxt - 1);
}
```

Actually, for this function you may choose to implement it by calling either `Level_drawColumn` or `Level_drawRow` under the hood, but here I'm going to implement it using `Level_drawColumn`. See if you can implement it yourself using `Level_drawRow` instead.

The first thing it does is convert the current camera's x and y positions from pixels into tiles, and stores them in the variables `cxt` and `cyt` respectively.

Next, refer to the second conceptual diagram for this section to see which columns we can draw "normally," meaning by passing the same `x` and `realX` value to the `Level_drawColumn` function (hint: columns 0 through 29). Even though columns 28 and 29 (blue in the diagram) are inside the hidden portion of VRAM, we can still index their VRAM column and the relative position of the tiles in the level array by adding one for each column beyond the camera's current position (represented by the dot in the diagram) that we move. That's why the `for` loop can go from `0` to `VRAM_TILES_H - 2`.

For the remaining two columns (green in the diagram), we still need to increment their indices into VRAM (`cxt + 30` and `cxt + 31`), but at the same time we need to reference columns from the level array that are backward in x, relative to the camera's current position, hence the `cxt - 2` and `cxt - 1`.

If you are confused as to how much each of those last two columns needs to look backward in the level array, refer back to the first conceptual diagram for this section where the viewport is centered, and match the column numbers up when moving to the second conceptual diagram. Now you understand why I drew both diagrams.

Also notice that we didn't have to deal with any VRAM wrapping, or worry about referencing positions outside the bounds of the level array here, because that's already been handled for us inside the `Level_drawColumn` function.

The only function we haven't defined yet is the `Camera_update` function which will scroll the screen, and may draw a row and/or column into the hidden portion of VRAM for us, so define that now:

```
void Camera_update(CAMERA *c)
{
  uint8_t prevX = Screen.scrollX;
  uint8_t prevY = Screen.scrollY;
  Screen.scrollX = (uint8_t)c->x;
  Screen.scrollY = (uint8_t)c->y;

  int16_t cxt = c->x / TILE_WIDTH;
  int16_t cyt = c->y / TILE_HEIGHT;

  // Have we scrolled past a tile boundary along X?
  if ((prevX & ~(TILE_WIDTH - 1)) != (Screen.scrollX & ~(TILE_WIDTH - 1))) {
    if ((uint8_t)(Screen.scrollX - prevX) < (uint8_t)(prevX - Screen.scrollX))
      Level_drawColumn(c->level, cxt + 29, cyt, cxt + 29);
    else
      Level_drawColumn(c->level, cxt + 30, cyt, cxt - 2);
  }
  // Have we scrolled past a tile boundary along Y?
  if ((prevY & ~(TILE_HEIGHT - 1)) != (Screen.scrollY & ~(TILE_HEIGHT - 1))) {
    if ((uint8_t)(Screen.scrollY - prevY) < (uint8_t)(prevY - Screen.scrollY))
      Level_drawRow(c->level, cxt, cyt + 29, cyt + 29);
    else
      Level_drawRow(c->level, cxt, cyt + 30, cyt - 2);
  }
}
```

Since this function is called every frame, before we scroll the screen to the current camera position (in pixels) by setting `Screen.scrollX` and `Screen.scrollY`, we store their previous values in `prevX` and `prevY` respectively.

Next we convert the current camera's x and y positions from pixels into tiles, storing them in `cxt` and `cyt` respectively for later use.

Then we check to see if we just scrolled past a horizontal tile boundary, which occurs every 8 pixels, by masking off the lowest three bits (forcing them to zero) from both `prevX` and `Screen.scrollX` and then comparing only their five highest bits with each other. If those high bits are the same, we haven't moved off the current tile yet, so we don't need to call `Level_drawColumn`, but if they are different then we know we just scrolled past a tile boundary and we need to refresh the data in a hidden column.[7]

If we crossed a horizontal tile boundary, we calculate which direction we just scrolled, and call `Level_drawColumn` on the column farthest away from the viewport in the direction we scrolled (column 29 or 30 in the conceptual diagrams) passing the appropriate value for `realX`.

After that, we do the exact same thing using `prevY` and `Screen.scrollY` to determine if we've just scrolled past a vertical tile boundary, determine the direction we scrolled, and then call `Level_drawRow` to draw the appropriate row into the hidden area of VRAM.

Be aware that this code assumes you aren't going to move the camera by more than 8 pixels in a given direction between calls to `Camera_update`, but that's a fair assumption for most games.

And that's everything you need to get bi-directional horizontal and vertical mode 3 scrolling working on the Uzebox—in under 200 lines of actual code!

Go to your project's `default` directory, type `make` and you should have a `horiz-vert.uze` file that you can run in either the `uzem` emulator, or the `cuzebox` emulator, or you can copy it to an SD card and run it on your real hardware. If you need help doing any of this, you can watch my video tutorial series.

A complete version of this project, including tile resources, and a `Makefile` is available online at:

https://github.com/artcfox/uzebox-mode-3-with-scrolling-guide

In order for the included `Makefile` to automatically find the Uzebox kernel and build tools, you should run:

```
git clone https://github.com/artcfox/uzebox-mode-3-with-scrolling-guide
```

from within your main uzebox directory, and not within an additional subdirectory, since this repository contains multiple projects that expect to find the kernel and build tools based on a relative path a specific number of directory levels away.

[7]Thanks to CunningFellow for this suggestion