**Smooth**

Example: Each entry in the matrix below represents a pixel, where means that the pixel’s red color value is , green is , blue is , and alpha is .

In C, we represent this matrix (2-D array) as a 1-D array using the RIDX function.

Thus, the matrix would have this representation in C as a 1-D array.

For smooth, we want to focus on the corner/edge cases, and then focus on the center.

The accumulate\_sum function takes in a pixel\_sum by reference and a pixel from .

For , the smooth function would call accumulate\_sum four times: one for each of the surrounding pixels and one for the pixel itself. So, we would make a new pixel\_sum, and pass that variable in by reference into accumulate\_sum along with the three surrounding pixels (, , and ) and the pixel itself (). Then, our pixel\_sum variable would look like this: . This is the sum, but we want the average. Luckily, pixel\_sum stores a num attribute (look at the struct for pixel\_sum!) that counts how many pixels we’ve summed together. To take the average, we simply take each of the four colors from the pixel\_sum variable and divide by num. Then, we set to equal this new pixel value.

It’s best to hard-code the corner/edge cases first, and then handle the center (since every pixel that’s not on the border will have all eight pixels surrounding it).

**Smooth**

Example: Each entry in the matrix below represents a pixel, where means that the pixel’s red color value is , green is , blue is , and alpha is .

In C, we represent this matrix (2-D array) as a 1-D array using the RIDX function.

Thus, the matrix would have this representation in C as a 1-D array.

For smooth, we want to focus on the corner/edge cases, and then focus on the center.

The accumulate\_sum function takes in a pixel\_sum by reference and a pixel from .

For , the smooth function would call accumulate\_sum four times: one for each of the surrounding pixels and one for the pixel itself. So, we would make a new pixel\_sum, and pass that variable in by reference into accumulate\_sum along with the three surrounding pixels (, , and ) and the pixel itself (). Then, our pixel\_sum variable would look like this: . This is the sum, but we want the average. Luckily, pixel\_sum stores a num attribute (look at the struct for pixel\_sum!) that counts how many pixels we’ve summed together. To take the average, we simply take each of the four colors from the pixel\_sum variable and divide by num. Then, we set to equal this new pixel value.

It’s best to hard-code the corner/edge cases first, and then handle the center (since every pixel that’s not on the border will have all eight pixels surrounding it).