Intermediate Programming

Day 16

Outline

- Exercise 15
- Midterm project

Exercise 15

Determine the endianness of the hardware, using the fact that:

950238851 = 0x38A37E83

⇒ Little endian

```
>> gcc endian.c ... -g
>> gdb ./a.out
...
(gdb) b main
...
(gdb) r
...
(gdb) n
...
(gdb) n
...
21          printf("%u\n", *p);
(gdb) print/x ((unsigned char *)p)[0]
$1 = 0x83
(gdb) print/x ((unsigned char *)p)[1]
$2 = 0x7e
(gdb) print/x ((unsigned char *)p)[2]
$3 = 0xa3
(gdb) print/x ((unsigned char *)p)[3]
$4 = 0x38

>>
```

Exercise 15

Implement **magnitude** without using signed integers.

interp.c

```
…
unsigned int magnitude( unsigned int value )
{
        if( value & (1<<31) ) return ~value+1;
        else return value;
}
…
```

Exercise 15

Implement **set_seed**.

random.c

```
…
void set_seed( int seed )
{
        srand( seed );
}
…
```

Exercise 15

Implement **gen_uniform**.

random.c

```
…
void set_seed( int seed )
{
        srand( seed );
}

int gen_uniform( int max_num )
{
        return rand() % max_num;
}
…
```

Exercise 15

Generate 500 uniformly distributed random numbers and increment the associated elements of the **hist** array.

random.c

```
…
int main( void )
{
        …
        for( unsigned int i=0 ; i<500 ; i++ ) hist[ gen_uniform(max_range) ]++;
        …
}
…
```

Exercise 15

Implement **normal_rand**.

random.c

```
…
void set_seed( int seed )
{
	srand( seed );
}

int gen_uniform( int max_num )
{
	return rand() % max_num;
}

int normal_rand( int max_num )
{
	const unsigned int N = 20;
	int sum = 0;
	for( unsigned int n=0 ; n<N ; n++ ) sum += gen_uniform( max_num );
	return sum / N;
}
```

Exercise 15

Generate 500 normally distributed random numbers and increment the associated elements of the **hist** array.

random.c

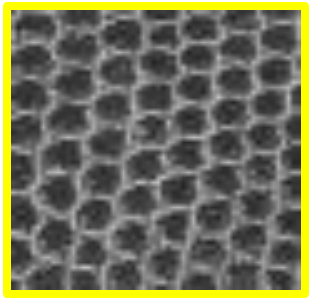
```
…
int main( void )
{
        …
        for( unsigned int i=0 ; i<500 ; i++ ) hist[ normal_rand(max_range) ]++;
        …
}
…
```

Outline

- Exercise 15
- Midterm project

Midterm project

exemplar



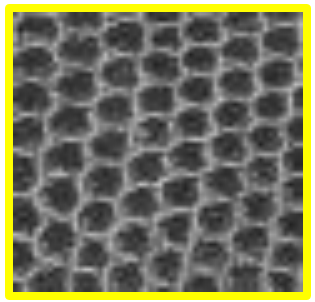
Goal:

Implement Efros and Leung's seminal work "Texture Synthesis by Non-parametric Sampling".

Given an exemplar texture image...

Midterm project

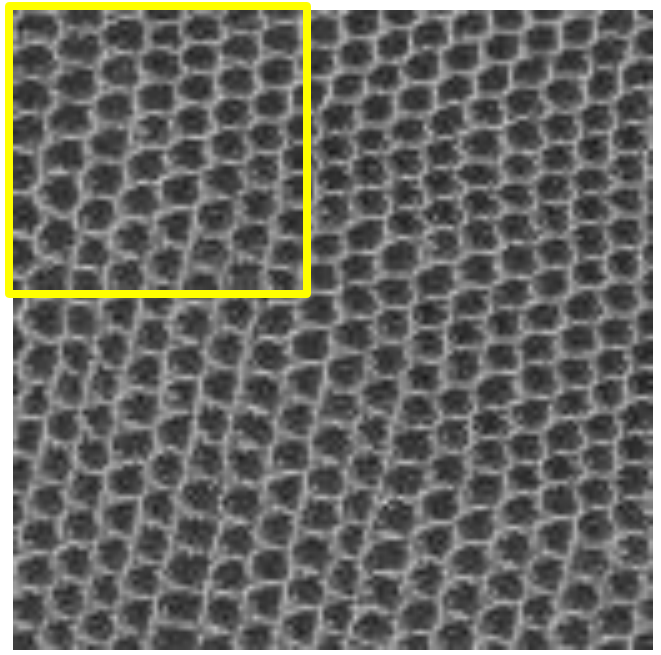
exemplar



Goal:

Implement Efros and Leung's seminal work "Texture Synthesis by Non-parametric Sampling".

Given an exemplar texture image...
synthesize a larger image by growing the texture.

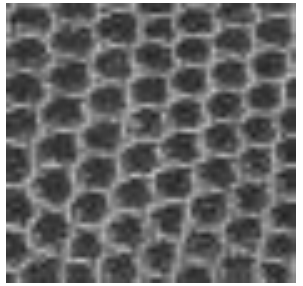


*synthesized**

*Recall that the pixel at (0,0) is the top left corner.

Midterm project

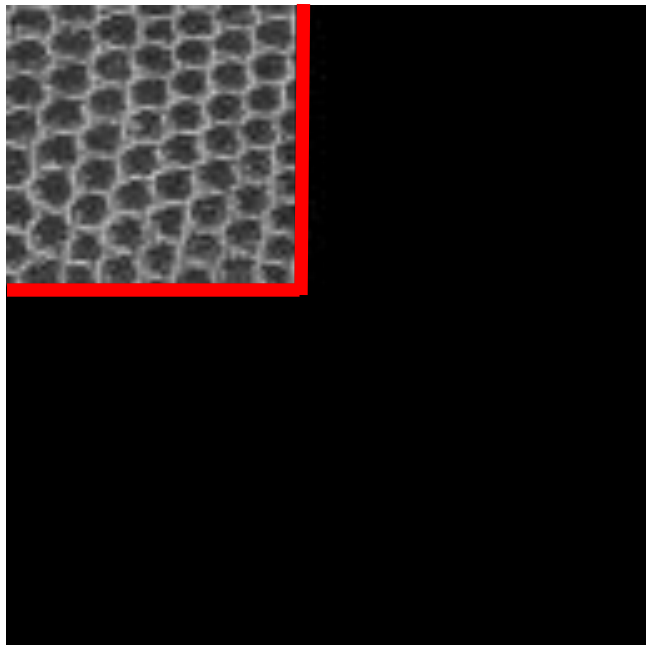
exemplar



Approach:

To expand the exemplar, identify all boundary pixels – pixels whose values have not been synthesized yet but whose neighbors have.

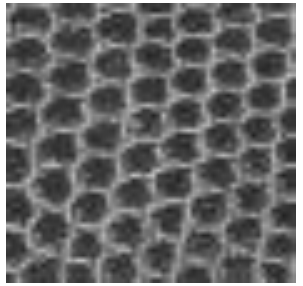
Assign those to-be-set pixels (TBS pixels) a color by copying good color values from the exemplar.



synthesized

Midterm project

exemplar

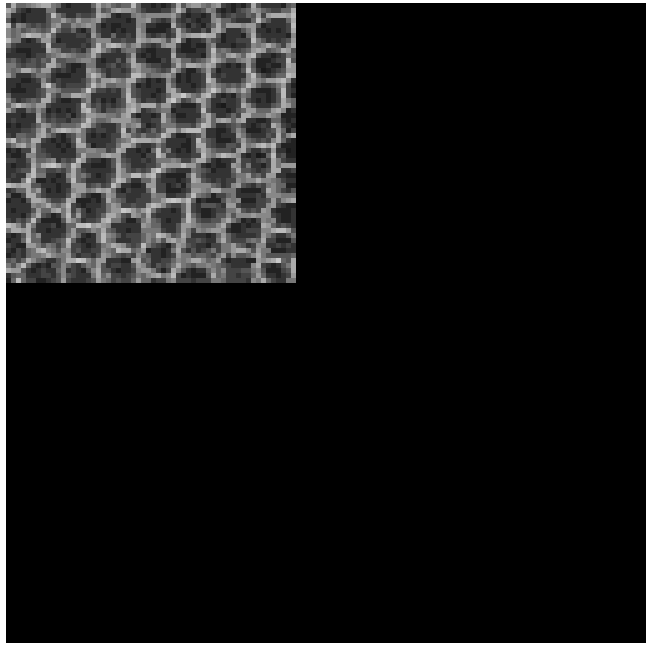


Approach:

To expand the exemplar, identify all boundary pixels – pixels whose values have not been synthesized yet but whose neighbors have.

Assign those to-be-set pixels (TBS pixels) a color by copying good color values from the exemplar.

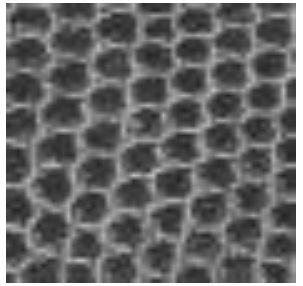
Repeat.



synthesized

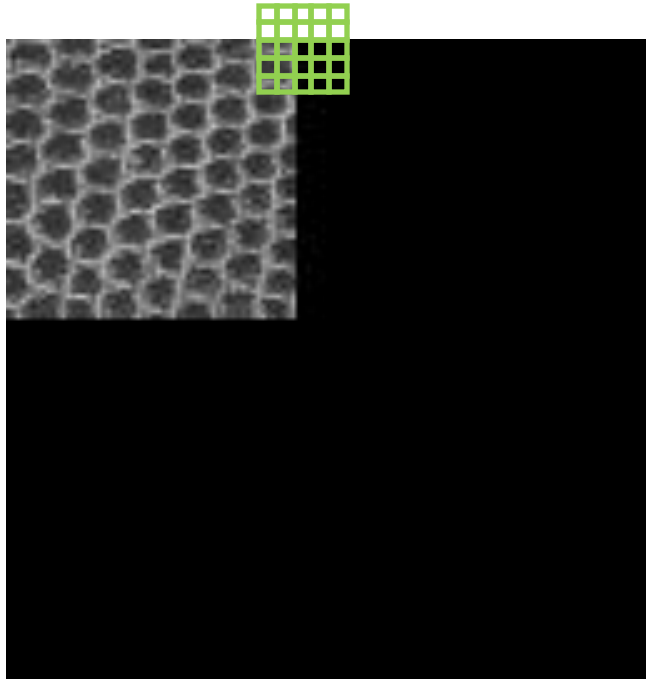
Midterm project

exemplar



Implementation (per TBS pixel):

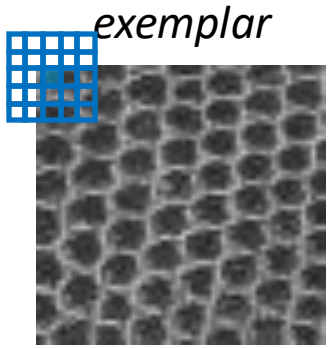
- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.



synthesized

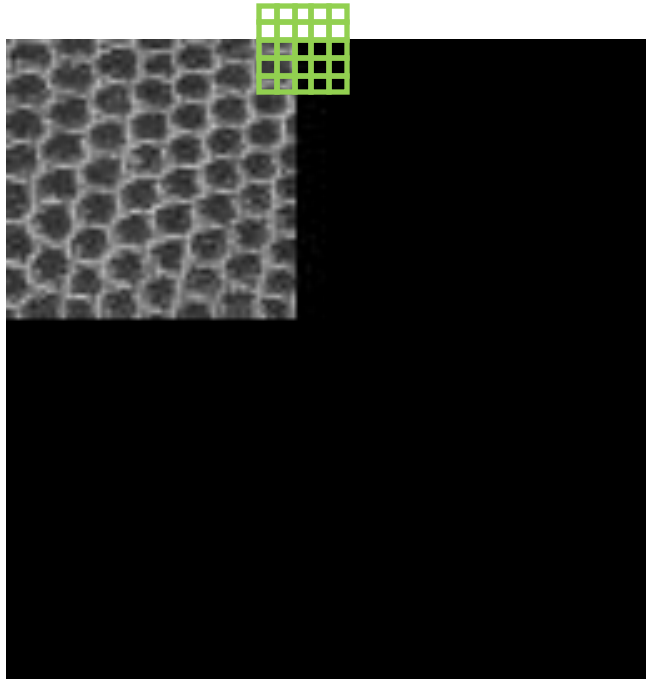
Visualization for $r = 2$.

Midterm project



Implementation (per TBS pixel):

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.



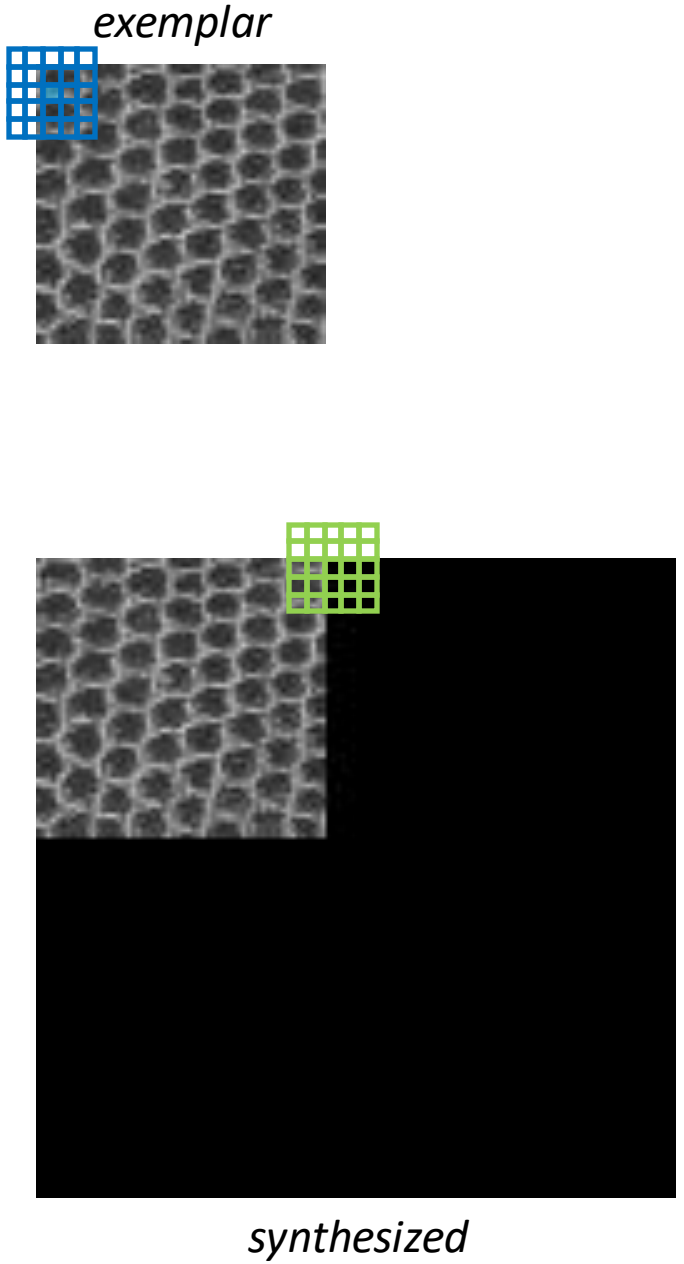
synthesized

Visualization for $r = 2$.

Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.



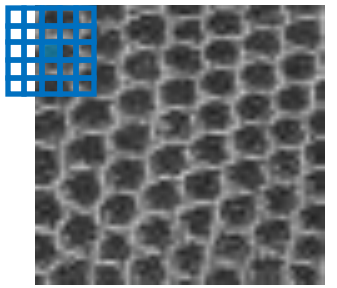
Visualization for $r = 2$.

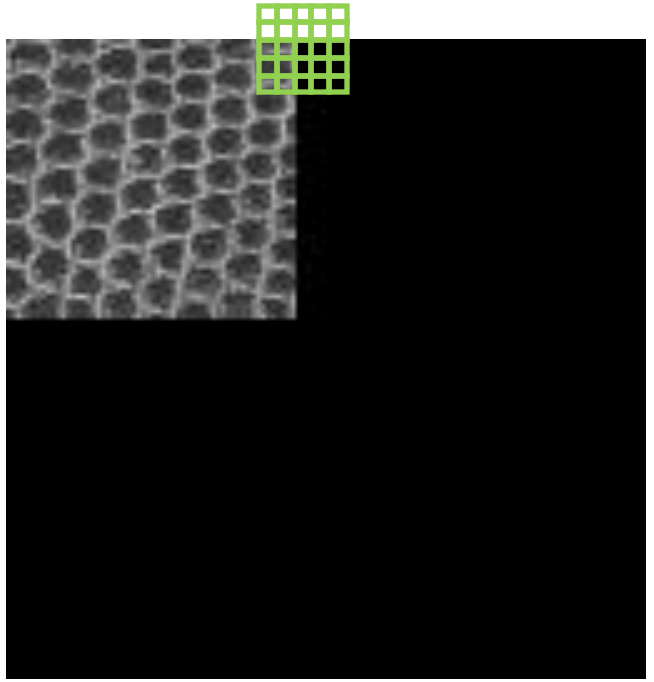
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

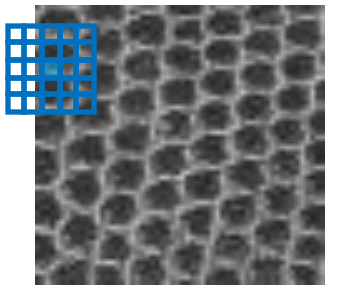
Visualization for $r = 2$.

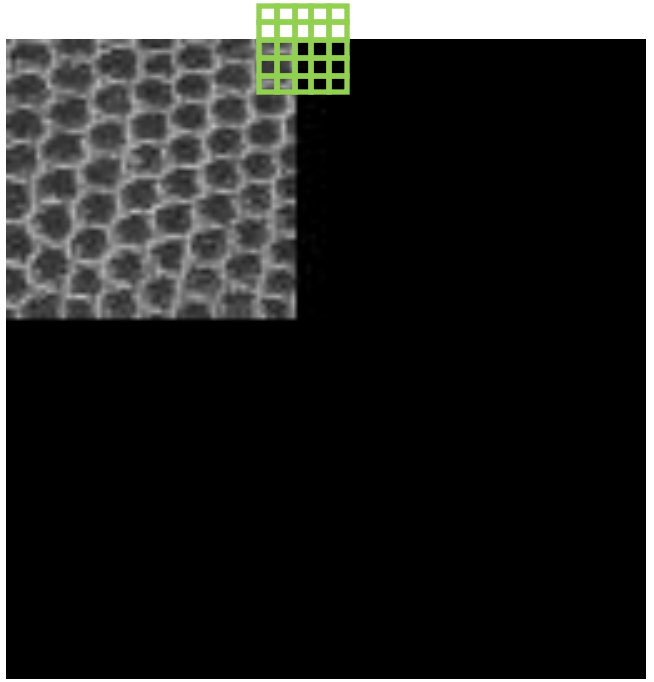
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar



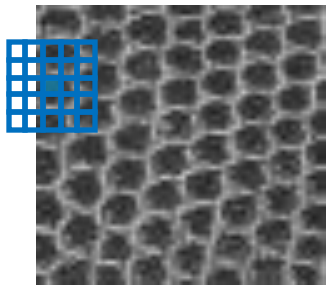


synthesized

Visualization for $r = 2$.

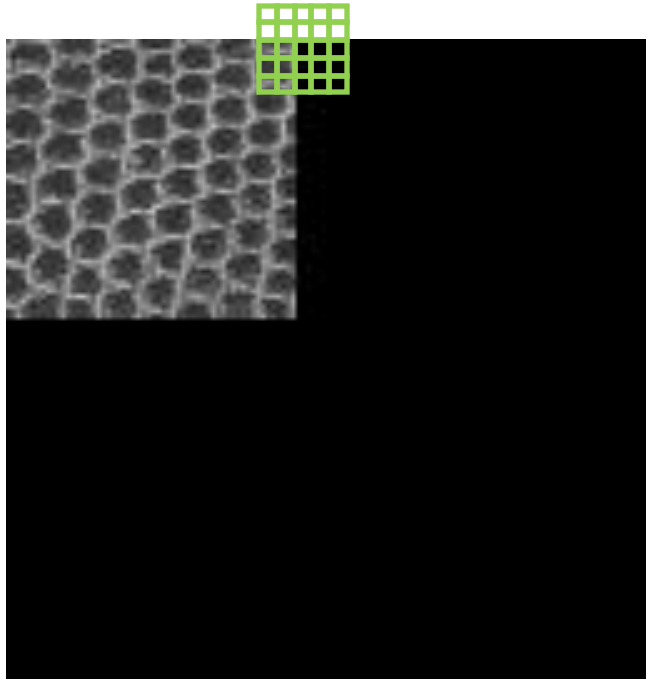
Midterm project

exemplar



Implementation (per TBS pixel):

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.



synthesized

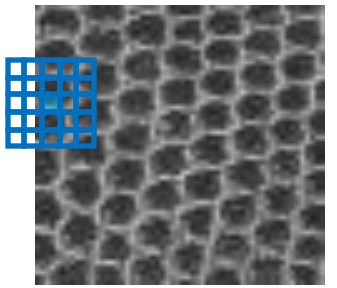
Visualization for $r = 2$.

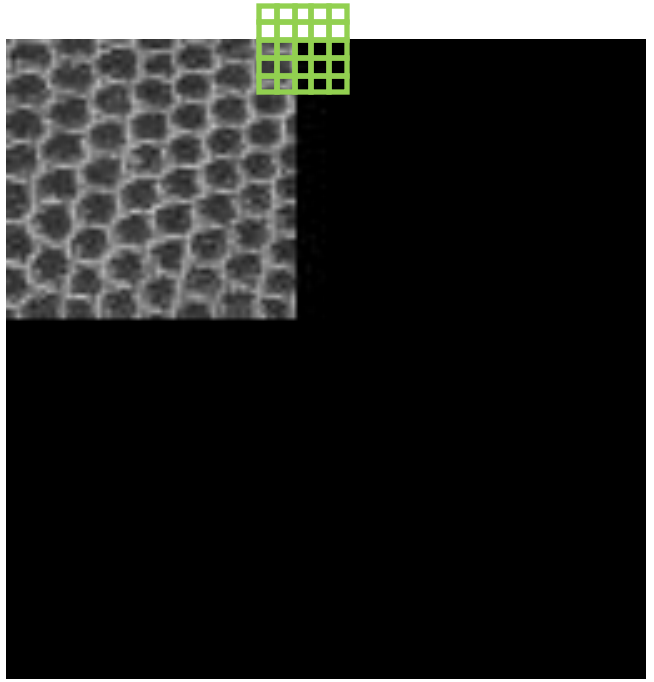
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

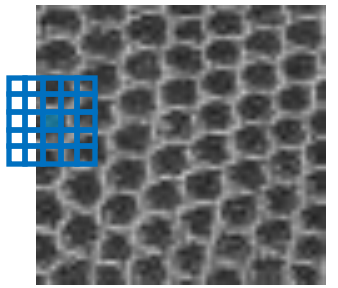
Visualization for $r = 2$.

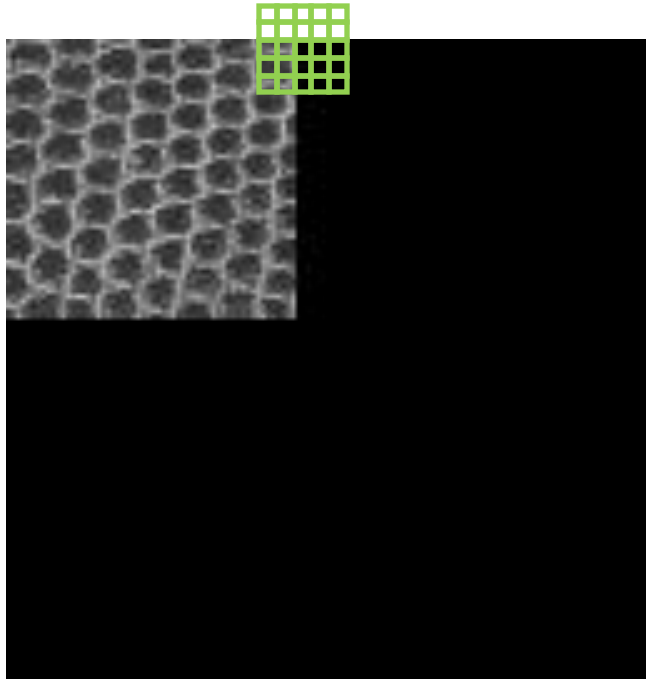
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

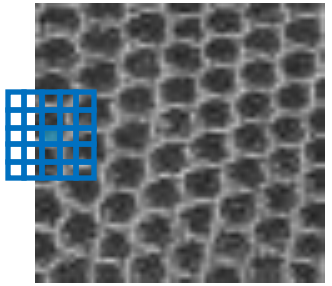
Visualization for $r = 2$.

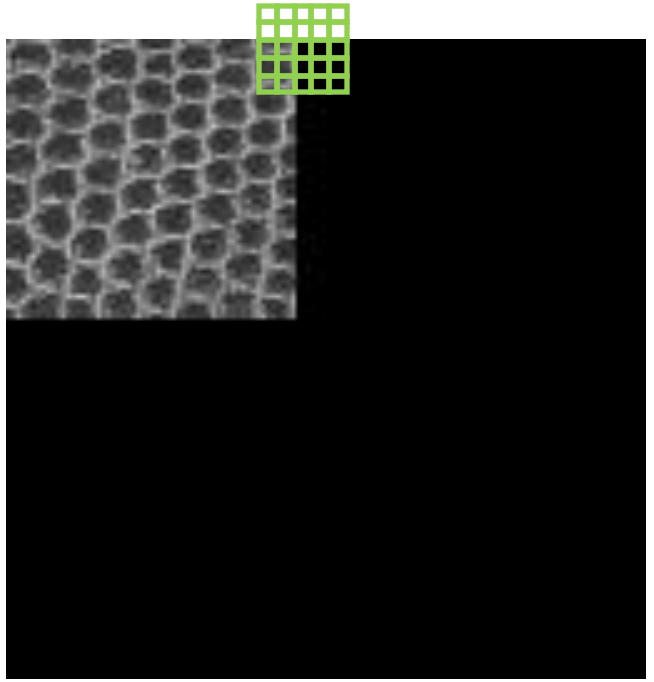
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

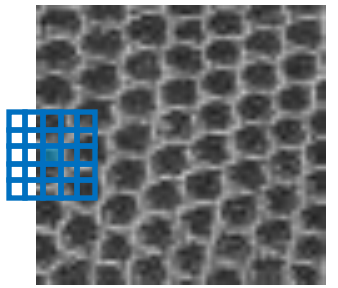
Visualization for $r = 2$.

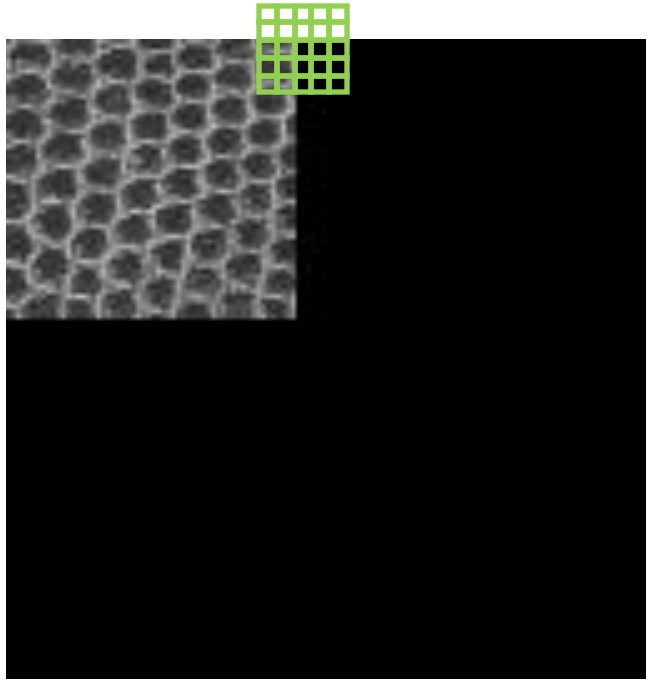
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

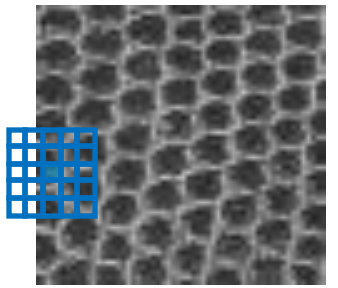
Visualization for $r = 2$.

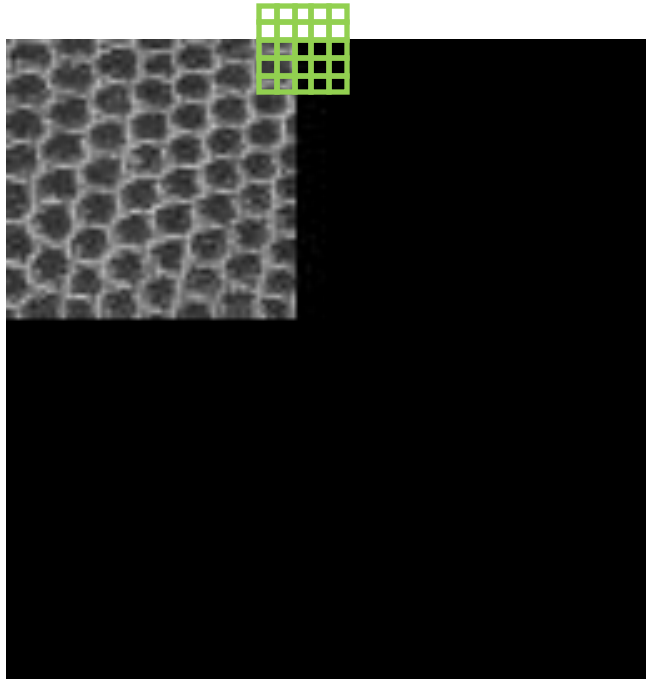
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

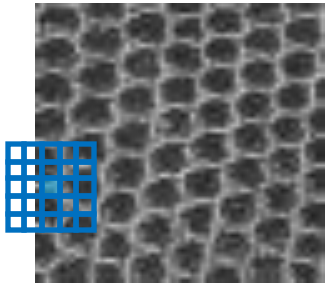
Visualization for $r = 2$.

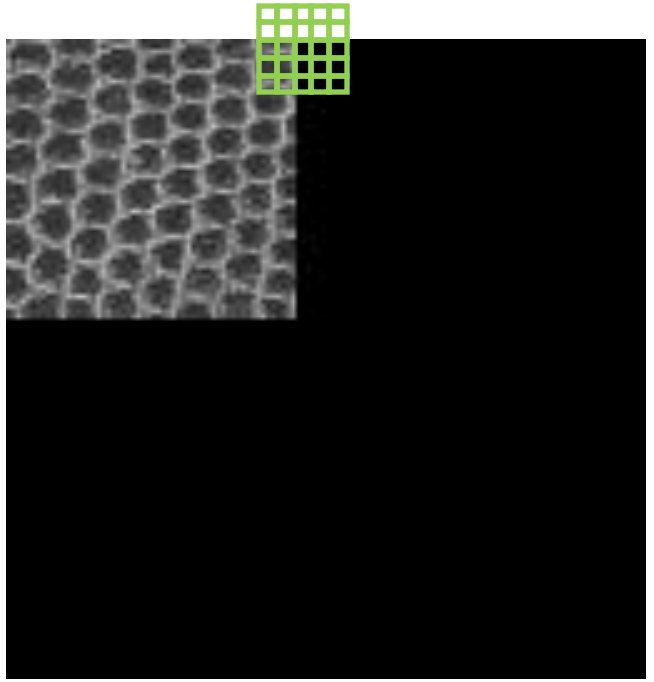
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

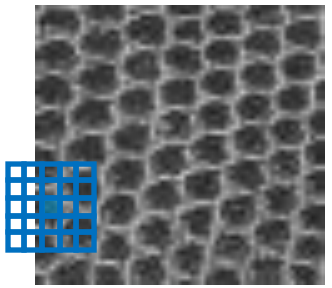
Visualization for $r = 2$.

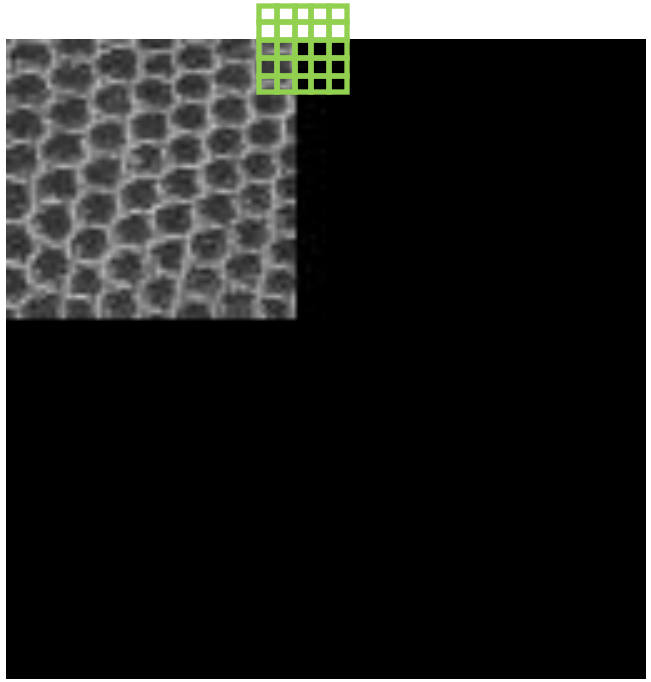
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

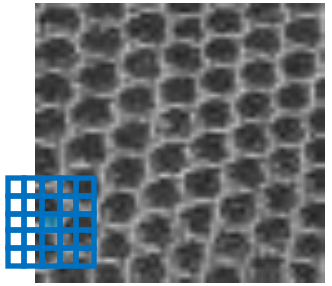
Visualization for $r = 2$.

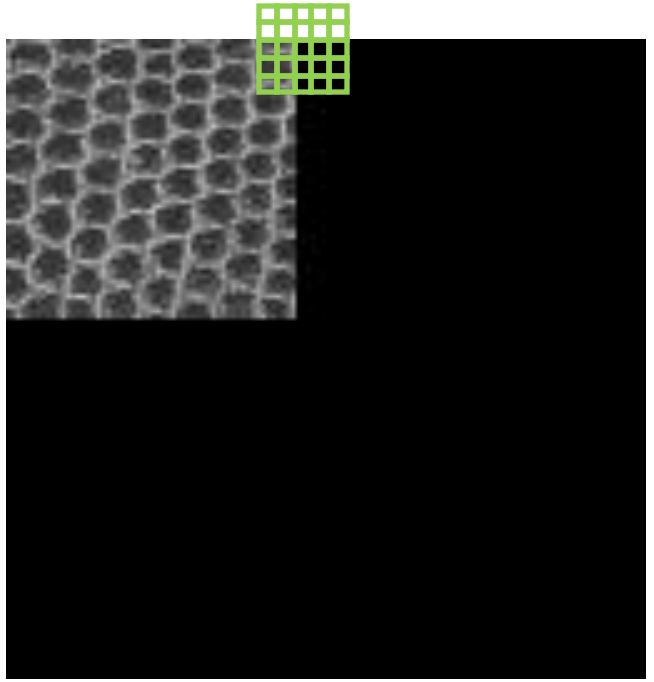
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

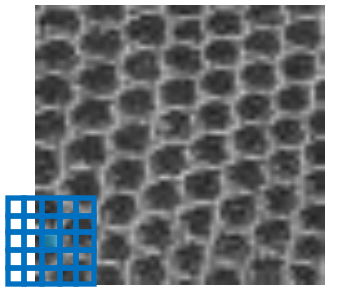
Visualization for $r = 2$.

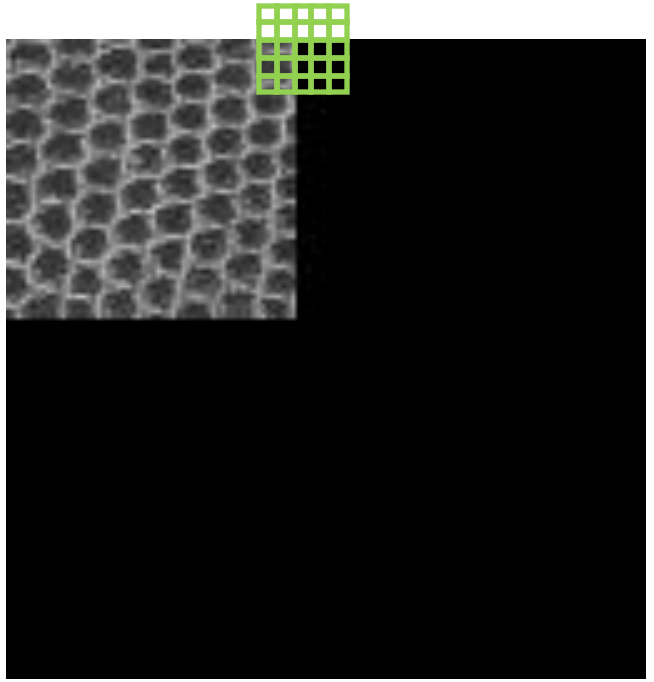
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

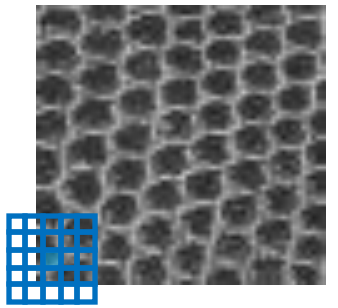
Visualization for $r = 2$.

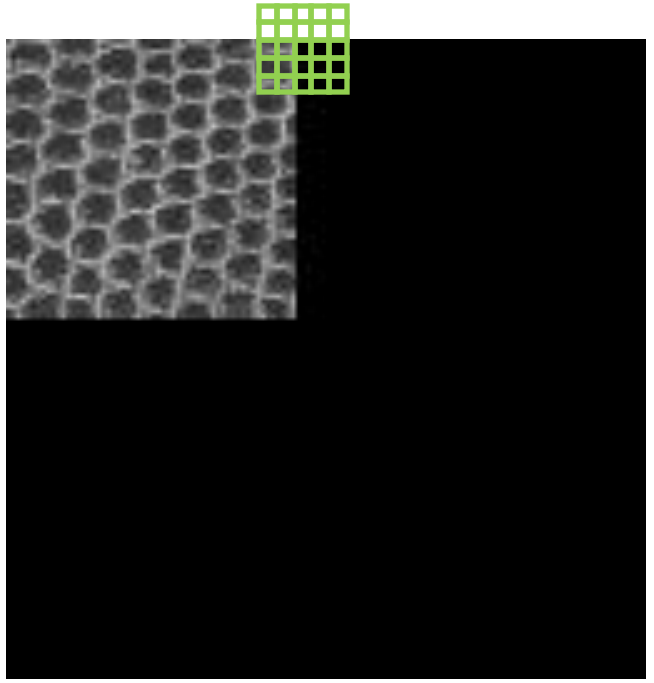
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

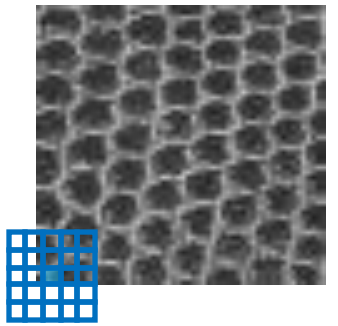
Visualization for $r = 2$.

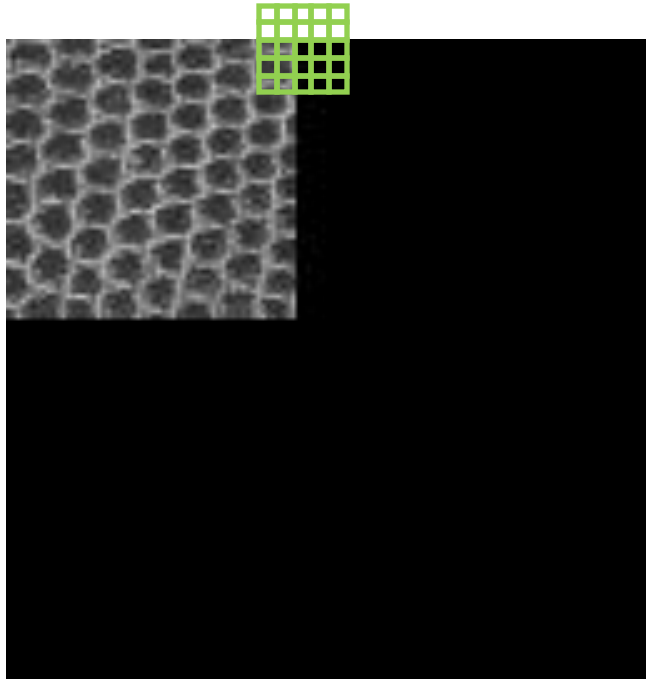
Midterm project

<u>Implementation (per TBS pixel)</u>:

- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.

exemplar





synthesized

Visualization for $r = 2$.

Midterm project

<u>Implementation (per TBS pixel)</u>:

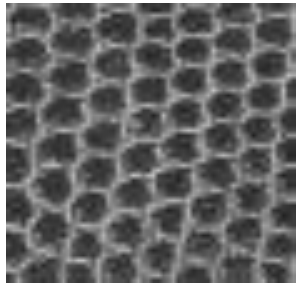
- Center a $(2r+1) \times (2r+1)$ window about the TBS pixel.
- For each exemplar pixel:
  - Measure how well the window about the exemplar pixel matches the window about the TBS pixel.
- Find the set of best matching exemplar pixels and select one at random from within the set.

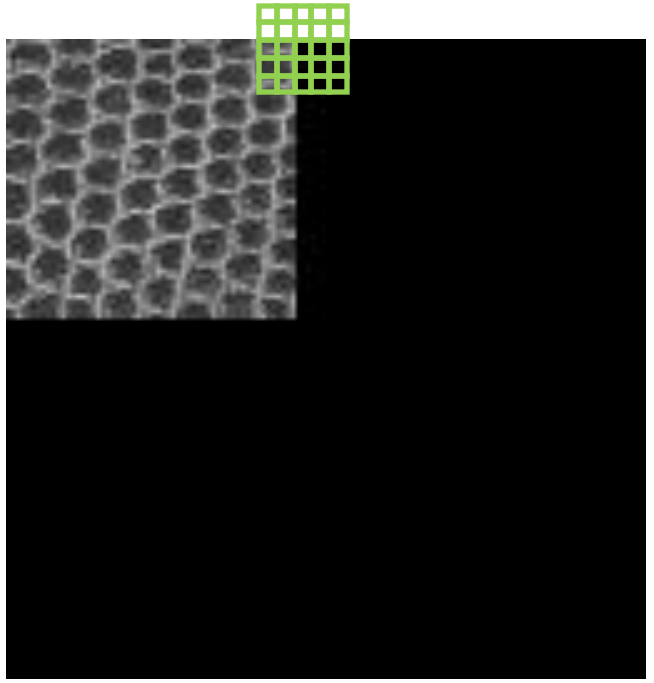
This is done for **every** TBS pixel.

Once you've processed the current set of TBS pixels, you need to generate the next set.

Visualization for $r = 2$.

exemplar

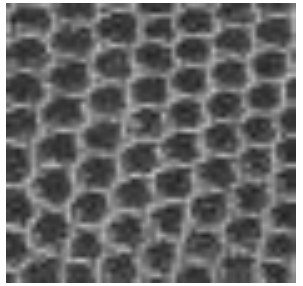




synthesized

Implementation details

exemplar

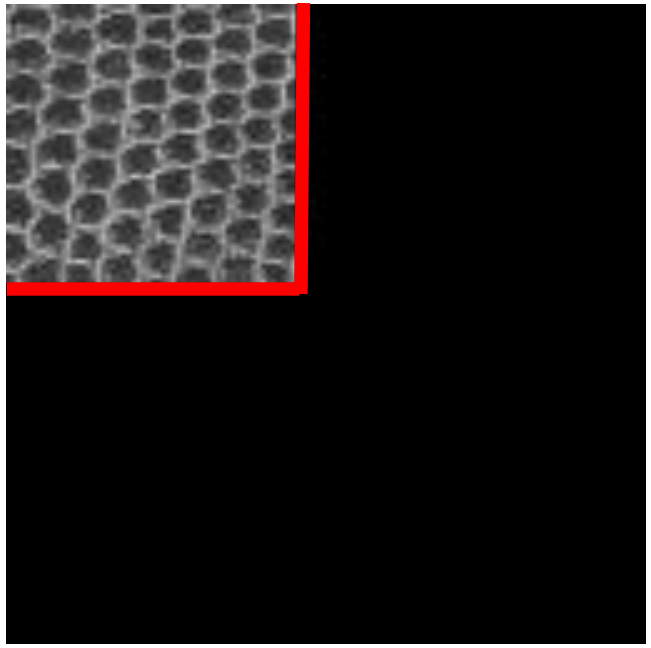


Processing the TBS pixels:

- You only want to process those pixels in the synthesized image that have not been *set* but have at least one (immediate) neighbor that has been *set*.
- When setting values of the current set of TBS pixels you want to set the values of those TBS pixels with more *set* neighbors first.

⇒ You will need to sort the array of current TBS pixels.

The function `SortTBSPixels` (in `image.[h/c]`) will help you with that.



synthesized

Aside

Sorting:

Because of the importance of sorting, C defines (`stdlib.h` declares) a function for sorting an array values from smallest to largest:

```
void qsort( void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *) );
```

Aside

Sorting:

Because of the importance of sorting, C defines (`stdlib.h` declares) a function for sorting an array values from smallest to largest:

```
void qsort( void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *) );
```

- `ptr`: a pointer to the first element in the array

Aside

Sorting:

Because of the importance of sorting, C defines (`stdlib.h` declares) a function for sorting an array values from smallest to largest:

```
void qsort( void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *) );
```

- `ptr`: a pointer to the first element in the array
- `count`: the number of elements in the array

Aside

Sorting:

Because of the importance of sorting, C defines (`stdlib.h` declares) a function for sorting an array values from smallest to largest:

```
void qsort( void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *) );
```

- `ptr`: a pointer to the first element in the array
- `count`: the number of elements in the array
- `size`: the size of an element

Aside

Sorting:

Because of the importance of sorting, C defines (`stdlib.h` declares) a function for sorting an array values from smallest to largest:

```
void qsort( void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *) );
```

- `ptr`: a pointer to the first element in the array
- `count`: the number of elements in the array
- `size`: the size of an element
- `cmp`: a pointer to a function taking two (`void`) pointers and returning an `int`
  - It returns a negative value if the first object pointed to is smaller than the second.
  - It returns a positive value if the first object pointed to is larger than the second.
  - It returns zero if they are the “same”.

Aside

Sorting:

Because of the importance of
function for sorting an array v

void qsort(void *ptr , size_t count ,

- ptr: a pointer to the first ele
- count: the number of eleme
- size: the size of an element
- cmp: a pointer to a function taking two (void) pointers and returning an int
  - It returns a negative value if the first object pointed to is smaller than the second.
  - It returns a positive value if the first object pointed to is larger than the second.
  - It returns zero if they are the "same".

sort_ints.c

```
#include <stdio.h>
#include <stdlib.h>

int cmp_ints( const void *v1 , const void *v2 ){ return *(int *)v2-*(int *)v1; }

int main( void )
{
	int a[5];
	for( unsigned int i=0 ; i<5 ; i++ ) a[i] = rand();

	printf( "Unsorted:\n" );
	for( unsigned int i=0 ; i<sz ; i++ ) printf( "\t %d] %d\n" , i , a[i] );

	qsort( a , 5 , sizeof(int) , cmp_ints );

	printf( "Sorted:\n" );
	for( unsigned int i=0 ; i<sz ; i++ ) printf( "\t %d] %d\n" , i , a[i] );

	return 0;
}
```

Aside

Sorting:

Because of the importance of
function for sorting an array v

void qsort(void *ptr , size_t count ,

- ptr: a pointer to the first ele
- count: the number of eleme
- size: the size of an element
- cmp: a pointer to a function taking two (void) pointers and
  - It returns a negative value if the first object pointed to is smaller th
  - It returns a positive value if the first object pointed to is larger tha
  - It returns zero if they are the "same".

sort_ints.c

```
#include <stdio.h>
#include <stdlib.h>

int cmp_ints( const void *v1 , const void *v2 ){ return *(int *)v2-*(int *)v1; }

int main( void )
{
	int a[5];
	for( unsigned int i=0 ; i<5 ; i++ ) a[i] = rand();

	printf( "Unsorted:\n" );
	for( unsigned int i=0 ; i<sz ; i++ ) pri

	qsort( a , 5 , sizeof(int) , cmp_ints

	printf( "Sorted:\n" );
	for( unsigned int i=0 ; i<sz ; i++ ) pri

	return 0;
}
```

```
>> ./a.out
Unsorted:
        0] 1804289383
        1] 846930886
        2] 1681692777
        3] 1714636915
        4] 1957747793
Sorted:
        0] 1957747793
        1] 1804289383
        2] 1714636915
        3] 1681692777
        4] 846930886
>>
```

Aside

Sorting:

Because of the importance of
function for sorting an array v

void qsort(void *ptr , size_t count ,

- ptr: a pointer to the first ele
- count: the number of eleme
- size: the size of an element
- cmp: a pointer to a function taking two (void) pointers and

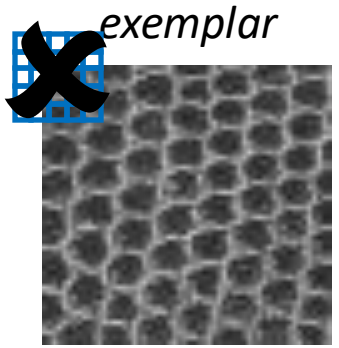
Note:
The sorted list is decreasing because cmp_ints returns the value of the second minus the value of the first.

sort_ints.c

```
#include <stdio.h>
#include <stdlib.h>

int cmp_ints( const void *v1 , const void *v2 ){ return *(int *)v2-*(int *)v1; }

int main( void )
{
	int a[5];
	for( unsigned int i=0 ; i<5 ; i++ ) a[i] = rand();

	printf( "Unsorted:\n" );
	for( unsigned int i=0 ; i<sz ; i++ ) pri

	qsort( a , 5 , sizeof(int) , cmp_ints

	printf( "Sorted:\n" );
	for( unsigned int i=0 ; i<sz ; i++ ) pri

	return 0;
}
```

```
>> ./a.out
Unsorted:
        0] 1804289383
        1] 846930886
        2] 1681692777
        3] 1714636915
        4] 1957747793
Sorted:
        0] 1957747793
        1] 1804289383
        2] 1714636915
        3] 1681692777
        4] 846930886
>>
```

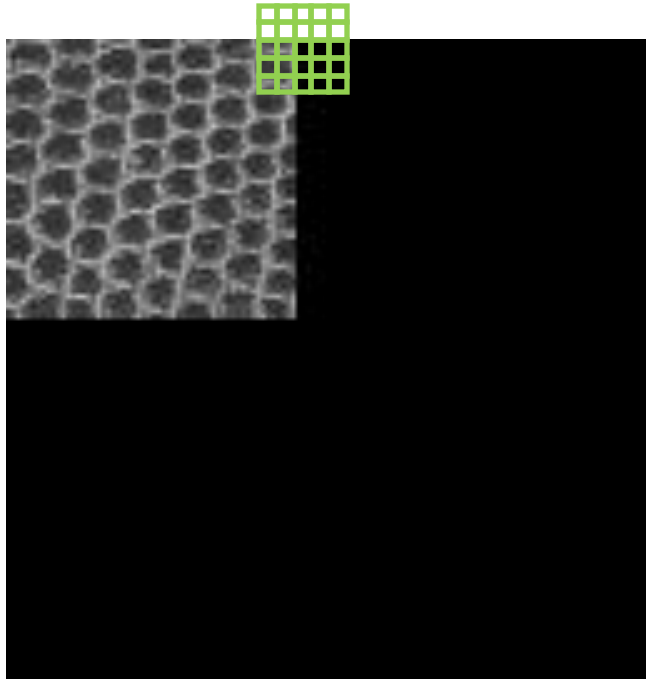
Implementation details



<u>Comparing TBS and exemplar windows</u>:

- You should only consider an exemplar pixel as a candidate match for a TBS pixel if:

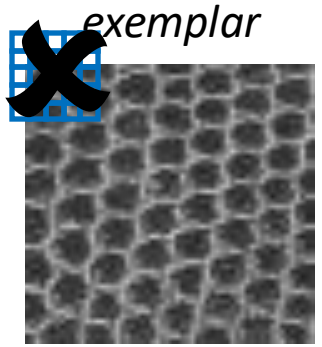
  For every *set* neighbor within the TBS window, the corresponding pixel about the exemplar pixel is within the exemplar image.



synthesized

Visualization for $r = 2$.

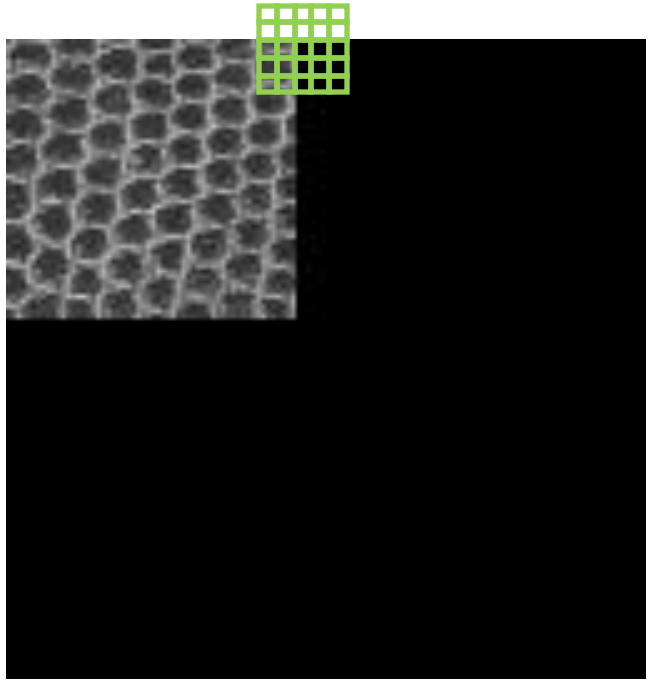
Implementation details



<u>Comparing TBS and exemplar windows</u>:

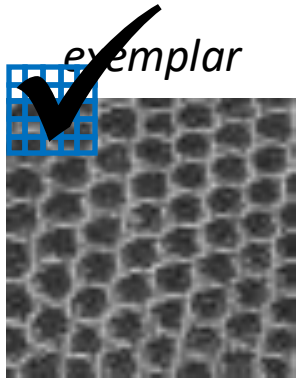
- You should only consider an exemplar pixel as a candidate match for a TBS pixel if:

  For every *set* neighbor within the TBS window, the corresponding pixel about the exemplar pixel is within the exemplar image.



Visualization for $r = 2$.

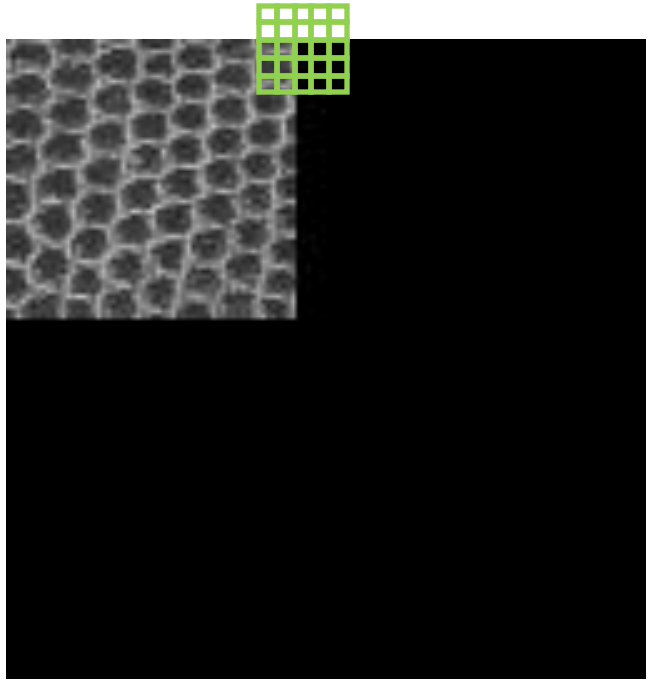
Implementation details



<u>Comparing TBS and exemplar windows</u>:

- You should only consider an exemplar pixel as a candidate match for a TBS pixel if:

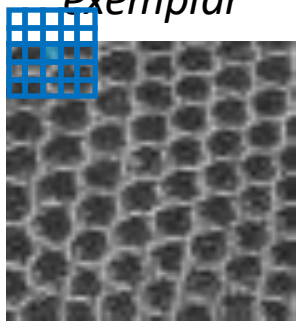
  For every *set* neighbor within the TBS window, the corresponding pixel about the exemplar pixel is within the exemplar image.



synthesized

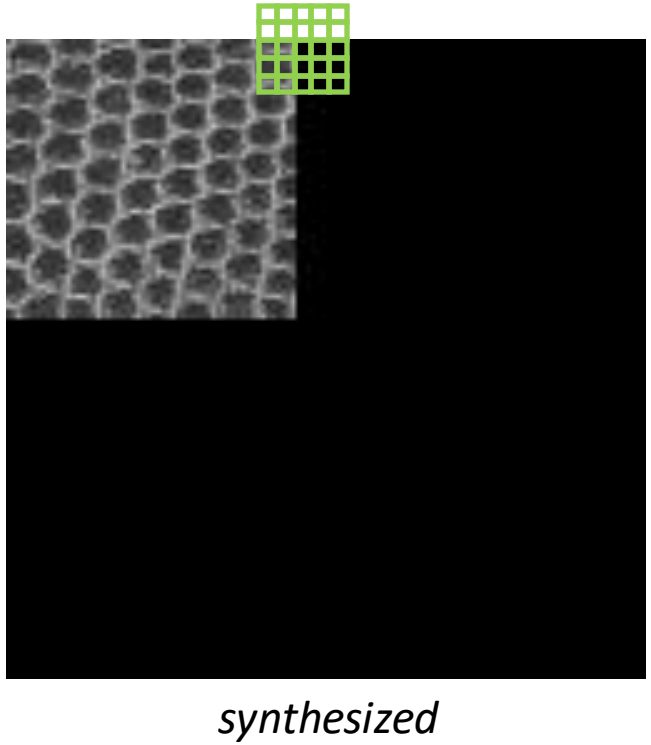
Visualization for $r = 2$.

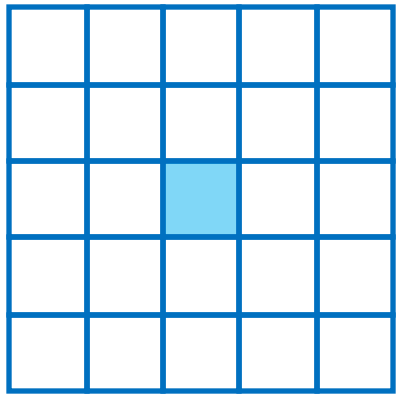
Implementation details

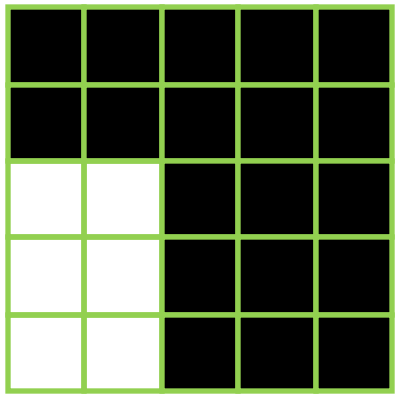


Comparing TBS and exemplar windows:

- For each pixel about the TBS pixel that is *set*
  - Sum the *weighted* squared differences of the red green, and blue values of corresponding pixels.

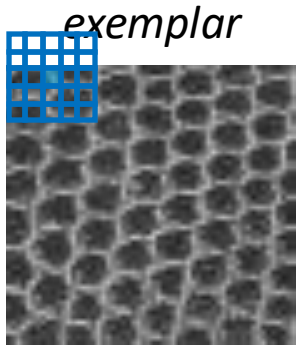






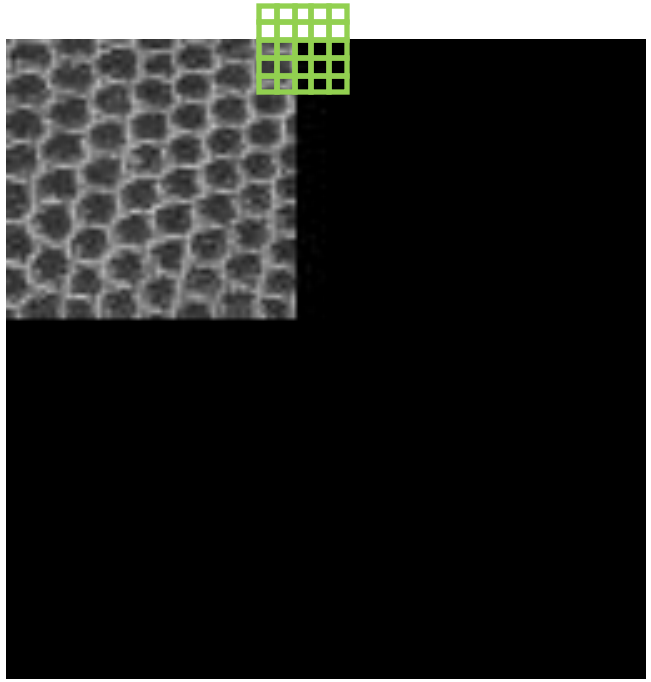
Visualization for $r = 2$.

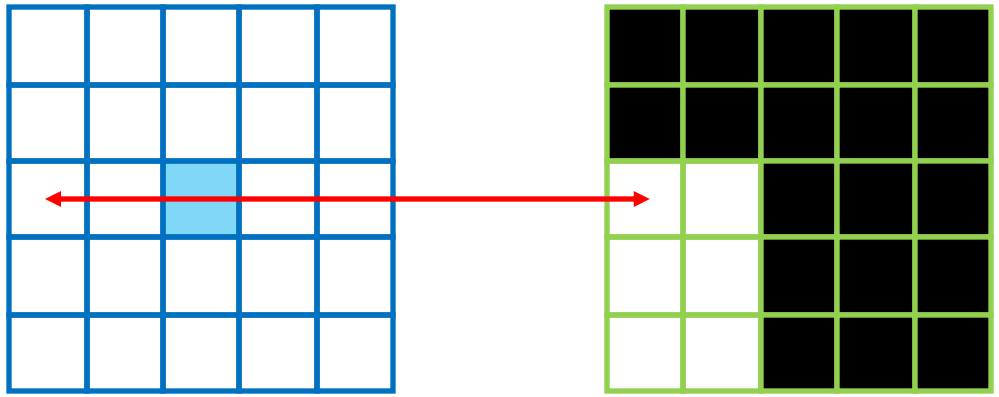
Implementation details



Comparing TBS and exemplar windows:

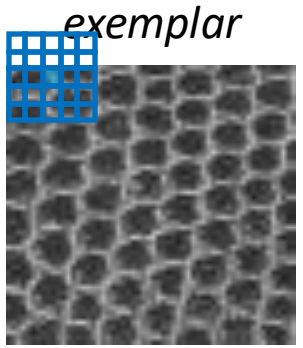
- For each pixel about the TBS pixel that is *set*
  - Sum the *weighted* squared differences of the red green, and blue values of corresponding pixels.





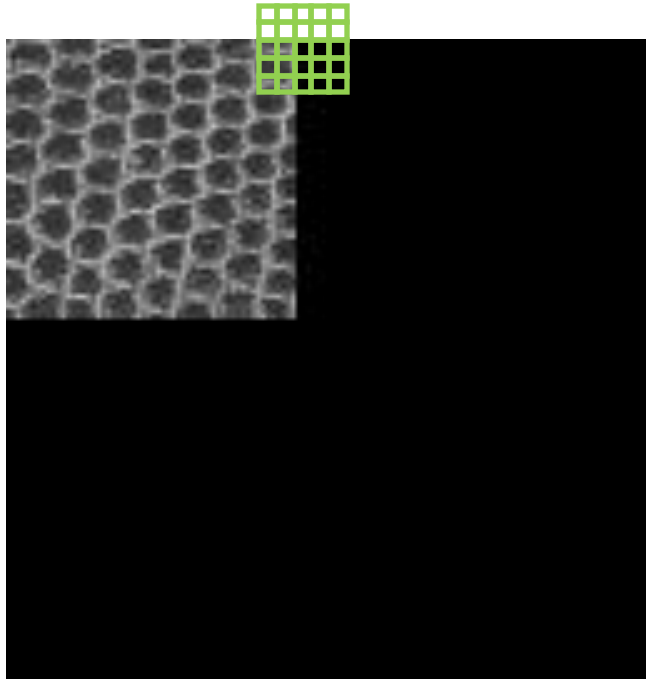
Visualization for $r = 2$.

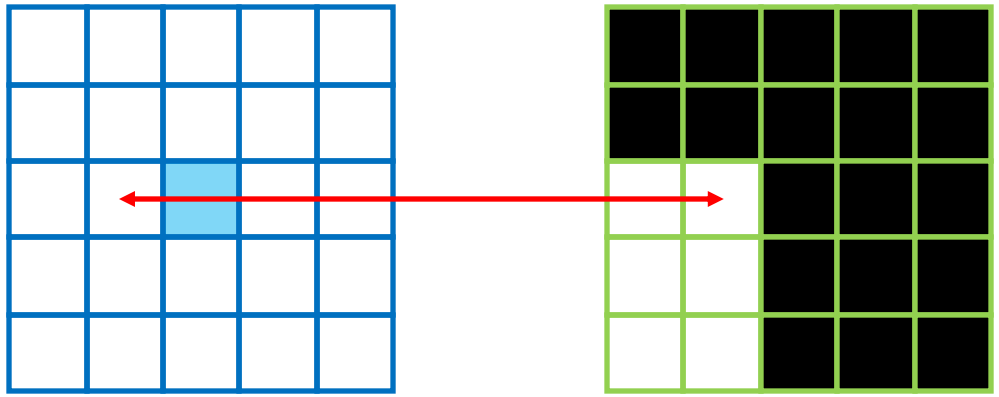
Implementation details



Comparing TBS and exemplar windows:

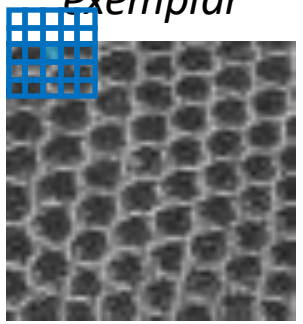
- For each pixel about the TBS pixel that is *set*
  - Sum the *weighted* squared differences of the red green, and blue values of corresponding pixels.





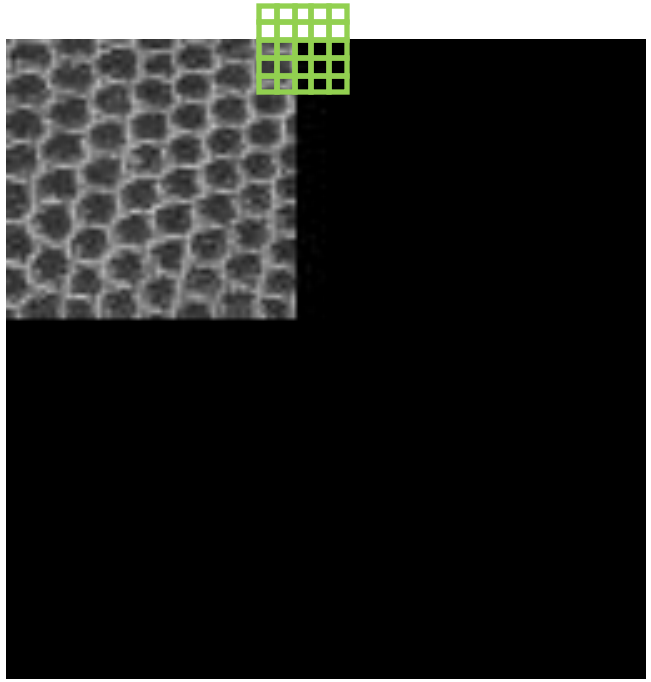
Visualization for $r = 2$.

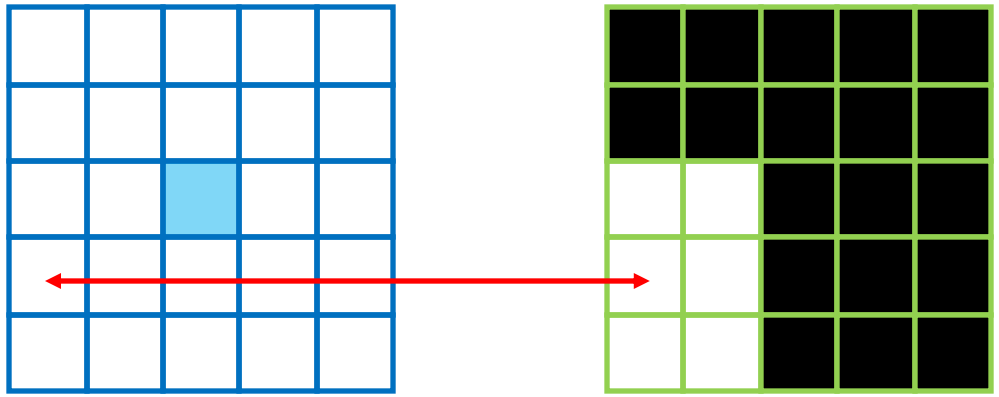
Implementation details



Comparing TBS and exemplar windows:

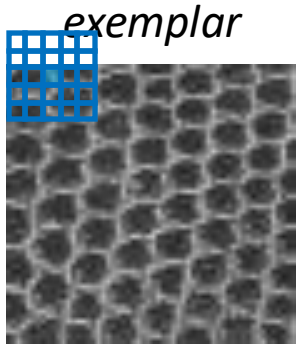
- For each pixel about the TBS pixel that is *set*
  - Sum the *weighted* squared differences of the red green, and blue values of corresponding pixels.





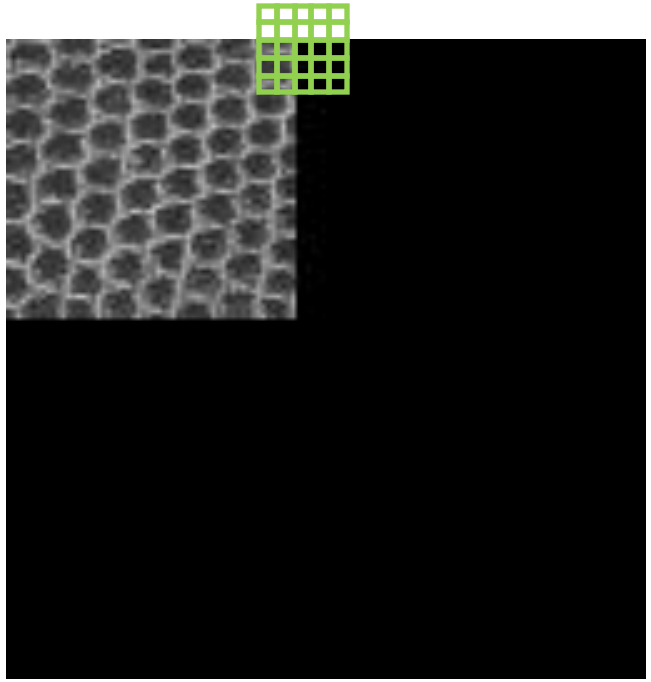
Visualization for $r = 2$.

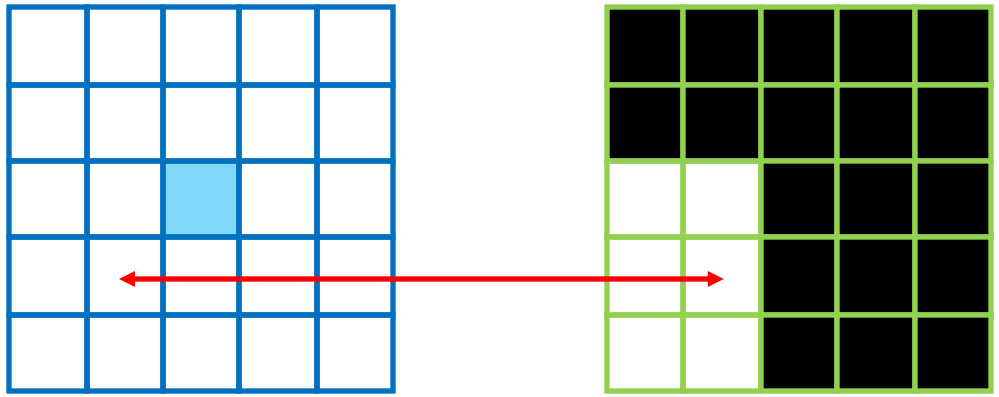
Implementation details



Comparing TBS and exemplar windows:

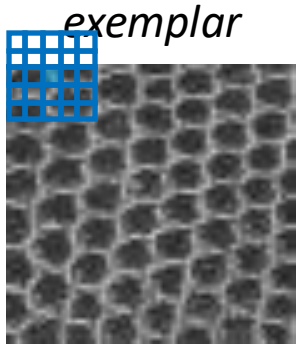
- For each pixel about the TBS pixel that is *set*
  - Sum the *weighted* squared differences of the red green, and blue values of corresponding pixels.





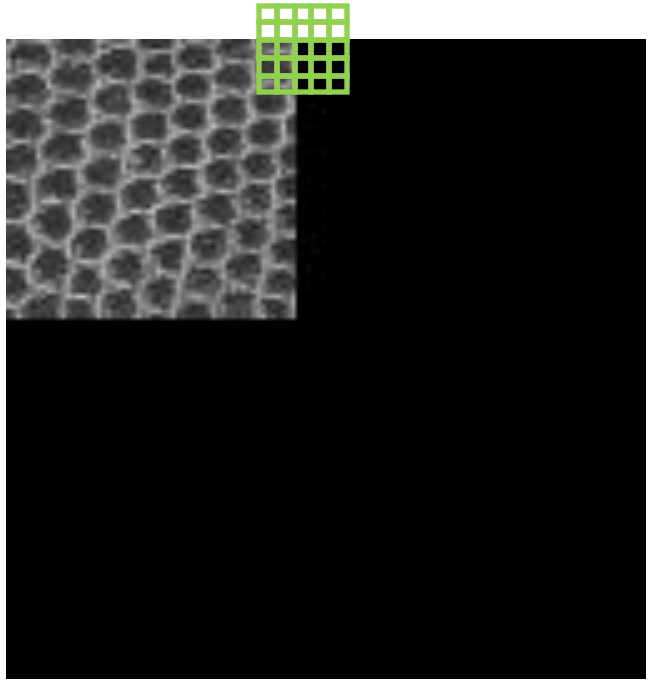
Visualization for $r = 2$.

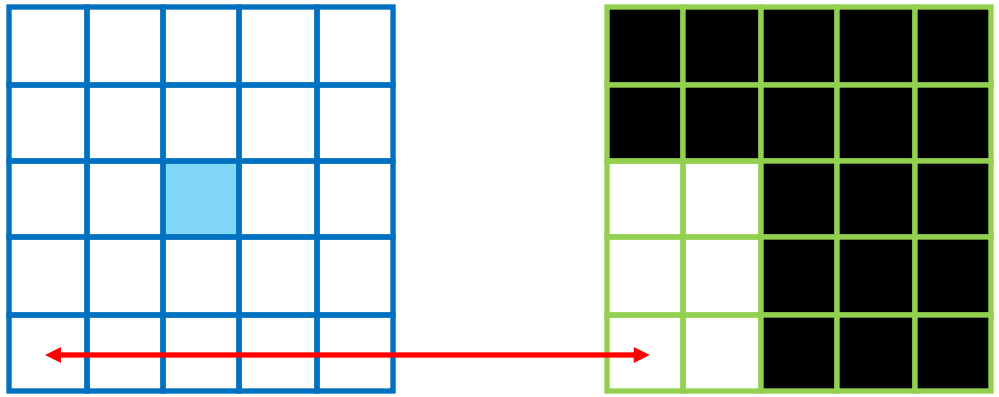
Implementation details



Comparing TBS and exemplar windows:

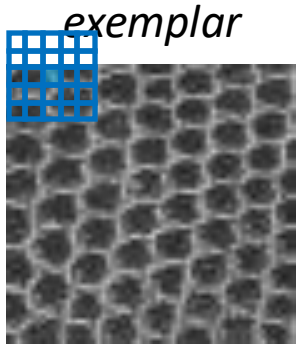
- For each pixel about the TBS pixel that is *set*
  - Sum the *weighted* squared differences of the red green, and blue values of corresponding pixels.





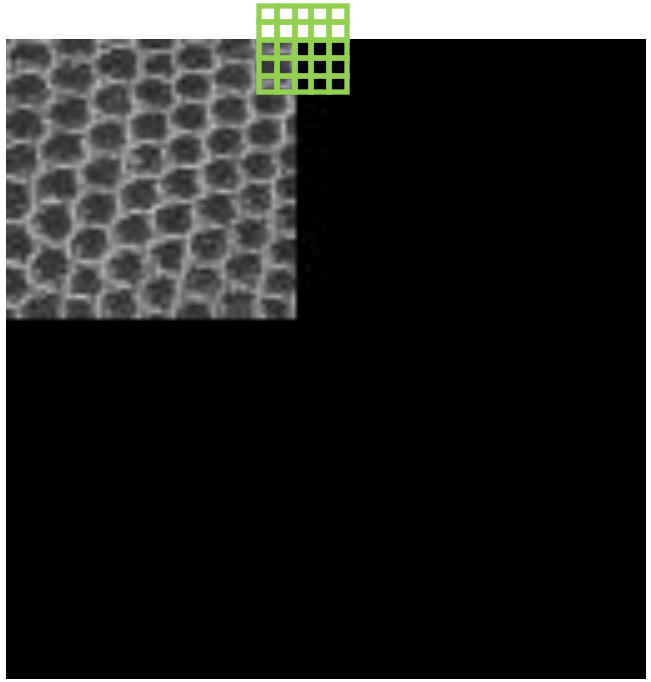
Visualization for $r = 2$.

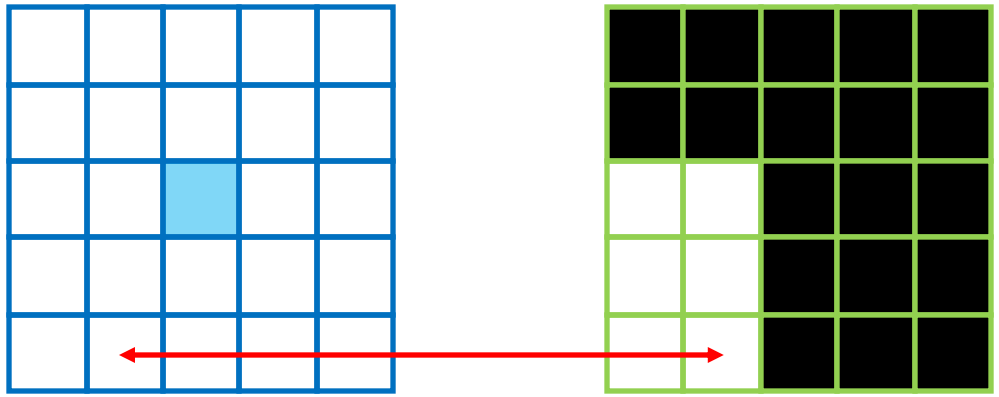
Implementation details



Comparing TBS and exemplar windows:

- For each pixel about the TBS pixel that is *set*
  - Sum the *weighted* squared differences of the red green, and blue values of corresponding pixels.





Visualization for $r = 2$.

Implementation details

Comparing TBS and exemplar windows:

- For each pixel about the TBS pixel that is *set*
  - Sum the *weighted* squared differences of the red green, and blue values of corresponding pixels.
  - The weight assigned to a pixel difference is given by a Gaussian with standard deviation $\sigma = \frac{2r+1}{6.4}$

$$w_{ij} = \exp\left(-\frac{i^2 + j^2}{2\sigma^2}\right)$$

where i and j are the horizontal and vertical coordinates of the pixel in the window relative to the center.

(-2,-2)	(-1,-2)	(0,-2)	(1,-2)	(2,-2)
(-2,-1)	(-1,-1)	(0,-1)	(1,-1)	(2,-1)
(-2,0)	(-1,0)	(0,0)	(1,0)	(2,0)
(-2,1)	(-1,1)	(0,1)	(1,1)	(2,1)
(-2,2)	(-1, 2)	(0, 2)	(1, 2)	(2, 2)

window indexing (i,j)

Visualization for $r = 2$.

Implementation details

Finding the set of best-matching exemplar pixels:

1. Identify the exemplar pixel minimizing the weighted sum of squared differences to the TBS pixel.
2. Let d_{min} be the weighted sum of squared difference for that exemplar pixel.
3. Define the set of best matching exemplar pixels to be those pixels whose weighted sum of squared differences to the TBS pixel is no larger than 1.1 times d_{min}.