Tarjan

Generated by Doxygen 1.9.6

1 Hierarchical Index 1

1.1 Class Hierarchy . . . 1

2 Class Index 3

2.1 Class List . . . 3

3 File Index 5

3.1 File List . . . 5

4 Class Documentation 7

4.1 Data Class Reference . . . 7

4.1.1 Detailed Description . . . 8

4.1.2 Constructor & Destructor Documentation . . . 8

4.1.2.1 Data() [1/7] . . . 8

4.1.2.2 Data() [2/7] . . . 8

4.1.2.3 Data() [3/7] . . . 9

4.1.2.4 Data() [4/7] . . . 9

4.1.2.5 Data() [5/7] . . . 9

4.1.2.6 Data() [6/7] . . . 10

4.1.2.7 Data() [7/7] . . . 10

4.1.2.8 ~Data() . . . 10

4.1.3 Member Function Documentation . . . 10

4.1.3.1 get_size() . . . 10

4.1.3.2 read_dir() . . . 11

4.1.3.3 read_file() [1/2] . . . 11

4.1.3.4 read_file() [2/2] . . . 11

4.1.3.5 set_input_file() . . . 11

4.1.3.6 update() . . . 12

4.1.3.7 write_file() [1/3] . . . 12

4.1.3.8 write_file() [2/3] . . . 12

4.1.3.9 write_file() [3/3] . . . 12

4.1.4 Member Data Documentation . . . 12

4.1.4.1 c . . . 13

4.1.4.2 data . . . 13

4.1.4.3 file_name . . . 13

4.1.4.4 file_read . . . 13

4.1.4.5 h . . . 13

4.1.4.6 input_file . . . 13

4.1.4.7 is_dir . . . 13

4.1.4.8 num_files . . . 13

4.1.4.9 size . . . 14

4.1.4.10 w . . . 14

4.2 Filter Class Reference . . . 14

4.2.1 Detailed Description . . . 14
4.2.2 Constructor & Destructor Documentation . . . 14
4.2.2.1 ~Filter() . . . 14
4.3 Image Class Reference . . . 15
4.3.1 Detailed Description . . . 16
4.3.2 Constructor & Destructor Documentation . . . 16
4.3.2.1 Image() [1/5] . . . 16
4.3.2.2 Image() [2/5] . . . 16
4.3.2.3 Image() [3/5] . . . 17
4.3.2.4 Image() [4/5] . . . 17
4.3.2.5 Image() [5/5] . . . 17
4.3.2.6 ~Image() . . . 17
4.3.3 Member Function Documentation . . . 17
4.3.3.1 get_channels() . . . 17
4.3.3.2 get_height() . . . 17
4.3.3.3 get_width() . . . 18
4.4 ImageFilter Class Reference . . . 18
4.4.1 Detailed Description . . . 19
4.4.2 Constructor & Destructor Documentation . . . 19
4.4.2.1 ~ImageFilter() . . . 19
4.4.3 Member Function Documentation . . . 19
4.4.3.1 adjust_brightness() . . . 19
4.4.3.2 BoxFilter() . . . 19
4.4.3.3 color_balance() . . . 20
4.4.3.4 edge_detection() . . . 20
4.4.3.5 GaussianFilter() . . . 20
4.4.3.6 histogram_equalization() . . . 21
4.4.3.7 median_filter() . . . 21
4.4.3.8 to_grayscale() . . . 21
4.5 Projection Class Reference . . . 21
4.5.1 Detailed Description . . . 22
4.5.2 Constructor & Destructor Documentation . . . 22
4.5.2.1 ~Projection() . . . 22
4.5.3 Member Function Documentation . . . 23
4.5.3.1 locating() . . . 23
4.5.3.2 max_intensity_projection() . . . 23
4.5.3.3 mean_avg_intensity_projection() . . . 23
4.5.3.4 median_avg_intensity_projection() . . . 24
4.5.3.5 min_intensity_projection() . . . 24
4.6 Slice Class Reference . . . 25
4.6.1 Detailed Description . . . 25
4.6.2 Member Function Documentation . . . 25

4.6.2.1 slice_xz() . . . 25
4.6.2.2 slice_yz() . . . 26
4.7 stbi_io_callbacks Struct Reference . . . 26
4.7.1 Member Data Documentation . . . 26
4.7.1.1 eof . . . 26
4.7.1.2 read . . . 27
4.7.1.3 skip . . . 27
4.8 Unittest Class Reference . . . 27
4.8.1 Detailed Description . . . 28
4.8.2 Member Function Documentation . . . 28
4.8.2.1 run_all() . . . 28
4.8.2.2 test_3D_GaussianFilter() . . . 28
4.8.2.3 test_3D_median() . . . 28
4.8.2.4 test_adjust_brightness() . . . 28
4.8.2.5 test_color_balance() . . . 29
4.8.2.6 test_edge_detection() . . . 29
4.8.2.7 test_GaussianFilter() . . . 29
4.8.2.8 test_GaussianFilter3d() . . . 29
4.8.2.9 test_histogram_equalization() . . . 29
4.8.2.10 test_mean_AIP() . . . 29
4.8.2.11 test_median_AIP() . . . 30
4.8.2.12 test_median_filter() . . . 30
4.8.2.13 test_MinIP() . . . 30
4.8.2.14 test_MIP() . . . 31
4.8.2.15 test_to_grayscale() . . . 31
4.9 Volume Class Reference . . . 32
4.9.1 Detailed Description . . . 33
4.9.2 Constructor & Destructor Documentation . . . 33
4.9.2.1 Volume() [1/3] . . . 33
4.9.2.2 Volume() [2/3] . . . 33
4.9.2.3 Volume() [3/3] . . . 34
4.9.2.4 ∼Volume() . . . 34
4.9.3 Member Function Documentation . . . 34
4.9.3.1 get_channels() . . . 34
4.9.3.2 get_x() . . . 34
4.9.3.3 get_y() . . . 34
4.9.3.4 get_z() . . . 34
4.9.4 Member Data Documentation . . . 34
4.9.4.1 dir_name . . . 35
4.10 VolumeFilter Class Reference . . . 35
4.10.1 Detailed Description . . . 35
4.10.2 Constructor & Destructor Documentation . . . 35

4.10.2.1 ~VolumeFilter() . . . 36
4.10.3 Member Function Documentation . . . 36
4.10.3.1 GaussianFilter3d() . . . 36
4.10.3.2 median3D_filter() . . . 36

5 File Documentation **37**
5.1 data/Data.cpp File Reference . . . 37
5.1.1 Macro Definition Documentation . . . 37
5.1.1.1 OUTPUT_DIR . . . 37
5.1.1.2 STB_IMAGE_IMPLEMENTATION . . . 37
5.1.1.3 STB_IMAGE_WRITE_IMPLEMENTATION . . . 38
5.2 data/Data.h File Reference . . . 38
5.3 Data.h . . . 38
5.4 filter/Filter.cpp File Reference . . . 39
5.5 filter/Filter.h File Reference . . . 39
5.6 Filter.h . . . 39
5.7 filter/ImageFilter.cpp File Reference . . . 39
5.8 filter/ImageFilter.h File Reference . . . 39
5.9 ImageFilter.h . . . 40
5.10 filter/VolumeFilter.cpp File Reference . . . 40
5.11 filter/VolumeFilter.h File Reference . . . 40
5.12 VolumeFilter.h . . . 41
5.13 image/Image.cpp File Reference . . . 41
5.13.1 Enumeration Type Documentation . . . 41
5.13.1.1 filter . . . 41
5.14 image/Image.h File Reference . . . 42
5.15 Image.h . . . 42
5.16 main.cpp File Reference . . . 42
5.16.1 Function Documentation . . . 43
5.16.1.1 is_int() . . . 43
5.16.1.2 main() . . . 43
5.17 minimal.cpp File Reference . . . 43
5.17.1 Macro Definition Documentation . . . 43
5.17.1.1 STB_IMAGE_IMPLEMENTATION . . . 43
5.17.1.2 STB_IMAGE_WRITE_IMPLEMENTATION . . . 44
5.17.2 Function Documentation . . . 44
5.17.2.1 main() . . . 44
5.18 projection/Projection.cpp File Reference . . . 44
5.19 projection/Projection.h File Reference . . . 44
5.20 Projection.h . . . 45
5.21 slice/Slice.cpp File Reference . . . 45
5.22 slice/Slice.h File Reference . . . 45

5.23 Slice.h . . . 45
5.24 stb/stb_image.h File Reference . . . 46
5.24.1 Macro Definition Documentation . . . 48
5.24.1.1 STBI_VERSION . . . 48
5.24.1.2 STBIDEF . . . 48
5.24.2 Typedef Documentation . . . 48
5.24.2.1 stbi_uc . . . 48
5.24.2.2 stbi_us . . . 48
5.24.3 Enumeration Type Documentation . . . 48
5.24.3.1 anonymous enum . . . 48
5.24.4 Function Documentation . . . 49
5.24.4.1 stbi_convert_iphone_png_to_rgb() . . . 49
5.24.4.2 stbi_convert_iphone_png_to_rgb_thread() . . . 49
5.24.4.3 stbi_failure_reason() . . . 49
5.24.4.4 stbi_hdr_to_ldr_gamma() . . . 49
5.24.4.5 stbi_hdr_to_ldr_scale() . . . 49
5.24.4.6 stbi_image_free() . . . 49
5.24.4.7 stbi_info() . . . 50
5.24.4.8 stbi_info_from_callbacks() . . . 50
5.24.4.9 stbi_info_from_file() . . . 50
5.24.4.10 stbi_info_from_memory() . . . 50
5.24.4.11 stbi_is_16_bit() . . . 50
5.24.4.12 stbi_is_16_bit_from_callbacks() . . . 51
5.24.4.13 stbi_is_16_bit_from_file() . . . 51
5.24.4.14 stbi_is_16_bit_from_memory() . . . 51
5.24.4.15 stbi_is_hdr() . . . 51
5.24.4.16 stbi_is_hdr_from_callbacks() . . . 51
5.24.4.17 stbi_is_hdr_from_file() . . . 51
5.24.4.18 stbi_is_hdr_from_memory() . . . 52
5.24.4.19 stbi_ldr_to_hdr_gamma() . . . 52
5.24.4.20 stbi_ldr_to_hdr_scale() . . . 52
5.24.4.21 stbi_load() . . . 52
5.24.4.22 stbi_load_16() . . . 52
5.24.4.23 stbi_load_16_from_callbacks() . . . 53
5.24.4.24 stbi_load_16_from_memory() . . . 53
5.24.4.25 stbi_load_from_callbacks() . . . 53
5.24.4.26 stbi_load_from_file() . . . 53
5.24.4.27 stbi_load_from_file_16() . . . 54
5.24.4.28 stbi_load_from_memory() . . . 54
5.24.4.29 stbi_load_gif_from_memory() . . . 54
5.24.4.30 stbi_loadf() . . . 54
5.24.4.31 stbi_loadf_from_callbacks() . . . 55

5.24.4.32 stbi_loadf_from_file() . . . 55
5.24.4.33 stbi_loadf_from_memory() . . . 55
5.24.4.34 stbi_set_flip_vertically_on_load() . . . 55
5.24.4.35 stbi_set_flip_vertically_on_load_thread() . . . 55
5.24.4.36 stbi_set_unpremultiply_on_load() . . . 56
5.24.4.37 stbi_set_unpremultiply_on_load_thread() . . . 56
5.24.4.38 stbi_zlib_decode_buffer() . . . 56
5.24.4.39 stbi_zlib_decode_malloc() . . . 56
5.24.4.40 stbi_zlib_decode_malloc_guesssize() . . . 56
5.24.4.41 stbi_zlib_decode_malloc_guesssize_headerflag() . . . 56
5.24.4.42 stbi_zlib_decode_noheader_buffer() . . . 57
5.24.4.43 stbi_zlib_decode_noheader_malloc() . . . 57
5.25 stb_image.h . . . 57
5.26 stb/stb_image_write.h File Reference . . . 151
5.26.1 Macro Definition Documentation . . . 152
5.26.1.1 STBIWDEF . . . 152
5.26.2 Typedef Documentation . . . 152
5.26.2.1 stbi_write_func . . . 152
5.26.3 Function Documentation . . . 152
5.26.3.1 stbi_flip_vertically_on_write() . . . 153
5.26.3.2 stbi_write_bmp() . . . 153
5.26.3.3 stbi_write_bmp_to_func() . . . 153
5.26.3.4 stbi_write_hdr() . . . 153
5.26.3.5 stbi_write_hdr_to_func() . . . 153
5.26.3.6 stbi_write_jpg() . . . 154
5.26.3.7 stbi_write_jpg_to_func() . . . 154
5.26.3.8 stbi_write_png() . . . 154
5.26.3.9 stbi_write_png_to_func() . . . 154
5.26.3.10 stbi_write_tga() . . . 155
5.26.3.11 stbi_write_tga_to_func() . . . 155
5.26.4 Variable Documentation . . . 155
5.26.4.1 stbi_write_force_png_filter . . . 155
5.26.4.2 stbi_write_png_compression_level . . . 155
5.26.4.3 stbi_write_tga_with_rle . . . 155
5.27 stb_image_write.h . . . 156
5.28 test.cpp File Reference . . . 177
5.28.1 Macro Definition Documentation . . . 177
5.28.1.1 OUTPUT_DIR . . . 178
5.28.1.2 STB_IMAGE_IMPLEMENTATION . . . 178
5.28.1.3 STB_IMAGE_WRITE_IMPLEMENTATION . . . 178
5.28.2 Function Documentation . . . 178
5.28.2.1 main() . . . 178

5.29 unittest/Unittest.cpp File Reference 178
5.30 unittest/Unittest.h File Reference 178
5.31 Unittest.h 179
5.32 volume/Volume.cpp File Reference 179
5.33 volume/Volume.h File Reference 179
5.34 Volume.h 180

Index **181**

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Data 7
- Image 15
- Volume 32

Filter 14
- ImageFilter 18
- Projection 21
- VolumeFilter 35

Slice 25
stbi_io_callbacks 26
Unittest 27

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Data 7
Filter 14
Image
    Class to represent 2D images 15
ImageFilter
    ImageFilter class which encapsulates filtering methods for 2D images 18
Projection
    Projection class which encapsulates projection methods for 3D volumes 21
Slice
    Slice class which encapsulates slice methods for 3D volumes 25
stbi_io_callbacks 26
Unittest 27
Volume
    Class to represent 3D volumes 32
VolumeFilter
    VolumeFilter class which encapsulates filtering methods for 3D volumes 35

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

main.cpp . . . 42
minimal.cpp . . . 43
test.cpp . . . 177
data/Data.cpp . . . 37
data/Data.h . . . 38
filter/Filter.cpp . . . 39
filter/Filter.h . . . 39
filter/ImageFilter.cpp . . . 39
filter/ImageFilter.h . . . 39
filter/VolumeFilter.cpp . . . 40
filter/VolumeFilter.h . . . 40
image/Image.cpp . . . 41
image/Image.h . . . 42
projection/Projection.cpp . . . 44
projection/Projection.h . . . 44
slice/Slice.cpp . . . 45
slice/Slice.h . . . 45
stb/stb_image.h . . . 46
stb/stb_image_write.h . . . 151
unittest/Unittest.cpp . . . 178
unittest/Unittest.h . . . 178
volume/Volume.cpp . . . 179
volume/Volume.h . . . 179

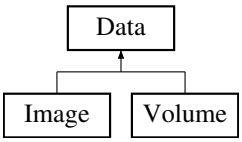
Chapter 4

Class Documentation

4.1 Data Class Reference

`#include <Data.h>`

Inheritance diagram for Data:



Public Member Functions

- Data ()
- Data (char *file_name)

  Constructor for Data used for single images. is_dir will default to False.
- Data (char *file_name, bool is_dir)

  Constructor for a new data object allowing the specification of is_dir. This is used for volume data, since an entire directory of images is read.
- Data (unsigned char *data, int w, int h, int c, char *name)

  Constructor for Data allowing the specification of a data array along with the dimensions of a single image. It is assumed to be a single image and is_dir will default to False.
- Data (unsigned char *data, int w, int h, int c, int num_files, char *name)

  Constructor for Data allowing the specification of a data array of many constituent images of a volume. It is assumed to be a volume and the maximum z dimension is num_files, the number of images.
- Data (Data &data)
- Data (Data &&data)
- virtual ~Data ()

  Destructor for Data. Ensures that allocated memory is freed.
- void read_dir ()

  Reads all images from a directory and stores it in the object's data variable representing a 3D volume.
- void read_file ()

  Reads the image file and stores it in data variable.
- void read_file (std::filesystem::path file, int channels)

Reads the image file and stores it in data variable.

- void write_file ()
- void write_file (std::string sub_dir)
- int get_size () const
- void set_input_file (char *file_name)

  Sets the input file of type filesystem::path to the file at file_name.
- void update (unsigned char *data, int channels, int size)

  Updates the object.

Static Public Member Functions

- static void write_file (unsigned char *write_data, int width, int height, int channels, std::string sub_dir, std::string file_name)

Public Attributes

- char * file_name
- std::filesystem::path input_file
- unsigned char * data
- int w
- int h
- int c
- int size
- int num_files
- bool file_read = false
- bool is_dir = false

4.1.1 Detailed Description

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.1.2 Constructor & Destructor Documentation

4.1.2.1 Data() [1/7]

```
Data::Data ( )
```

4.1.2.2 Data() [2/7]

```
Data::Data (
            char * file_name )
```

Constructor for Data used for single images. is_dir will default to False.

Parameters

file_name	The path to the image file.

4.1.2.3 Data() [3/7]

```
Data::Data (
            char * file_name,
            bool is_dir )
```

Constructor for a new data object allowing the specification of is_dir. This is used for volume data, since an entire directory of images is read.

Parameters

file_name	If is_dir is true, this is the path to a directory. Otherwise it is the path to a file.
is_dir	True if the data is a directory of images. False if the data is a single image.

4.1.2.4 Data() [4/7]

```
Data::Data (
            unsigned char * data,
            int w,
            int h,
            int c,
            char * name )
```

Constructor for Data allowing the specification of a data array along with the dimensions of a single image. It is assumed to be a single image and is_dir will default to False.

Parameters

data	The data array containing the image data.
w	The width of the image.
h	The height of the image.
c	The number of channels in the image.
name	The name of the image.

4.1.2.5 Data() [5/7]

```
Data::Data (
            unsigned char * data,
```

```
            int w,
            int h,
            int c,
            int num_files,
            char * name )
```

Constructor for Data allowing the specification of a data array of many constituent images of a volume. It is assumed to be a volume and the maximum z dimension is num_files, the number of images.

Parameters

data	The data array containing the volume data.
w	The width of a single image.
h	The height of a single image.
c	The number of channels in the image.
num_files	The number of constituent images.
name	The name of the volume.

4.1.2.6 Data() [6/7]

```
Data::Data (
            Data & data )
```

4.1.2.7 Data() [7/7]

```
Data::Data (
            Data && data )
```

4.1.2.8 ~Data()

```
Data::~Data ( )  [virtual]
```

Destructor for Data. Ensures that allocated memory is freed.

4.1.3 Member Function Documentation

4.1.3.1 get_size()

```
int Data::get_size ( ) const
```

4.1.3.2 read_dir()

```
void Data::read_dir ( )
```

Reads all images from a directory and stores it in the object's data variable representing a 3D volume.

4.1.3.3 read_file() [1/2]

```
void Data::read_file ( )
```

Reads the image file and stores it in data variable.

Returns

void

4.1.3.4 read_file() [2/2]

```
void Data::read_file (
            std::filesystem::path file,
            int channels )
```

Reads the image file and stores it in data variable.

Returns

void

4.1.3.5 set_input_file()

```
void Data::set_input_file (
            char * file_name )
```

Sets the input file of type filesystem::path to the file at file_name.

Parameters

file_name	The path to the file we wish to set as the input_file class variable

Returns

void

4.1.3.6 update()

```
void Data::update (
            unsigned char * new_data,
            int new_channels,
            int new_size )
```

Updates the object.

Parameters

new_data	
new_channels	
new_size	

Returns

void

4.1.3.7 write_file() [1/3]

```
void Data::write_file ( )
```

4.1.3.8 write_file() [2/3]

```
void Data::write_file (
            std::string sub_dir )
```

4.1.3.9 write_file() [3/3]

```
void Data::write_file (
            unsigned char * write_data,
            int width,
            int height,
            int channels,
            std::string sub_dir,
            std::string file_name )  [static]
```

4.1.4 Member Data Documentation

4.1.4.1 c

```
int Data::c
```

4.1.4.2 data

```
unsigned char* Data::data
```

4.1.4.3 file_name

```
char* Data::file_name
```

4.1.4.4 file_read

```
bool Data::file_read = false
```

4.1.4.5 h

```
int Data::h
```

4.1.4.6 input_file

```
std::filesystem::path Data::input_file
```

4.1.4.7 is_dir

```
bool Data::is_dir = false
```

4.1.4.8 num_files

```
int Data::num_files
```

4.1.4.9 size

```
int Data::size
```

4.1.4.10 w

```
int Data::w
```

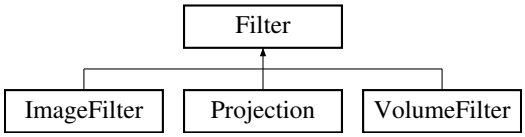
The documentation for this class was generated from the following files:

- data/Data.h
- data/Data.cpp

4.2 Filter Class Reference

```
#include <Filter.h>
```

Inheritance diagram for Filter:



Public Member Functions

- virtual ~Filter ()

4.2.1 Detailed Description

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322 Implementation of a Filter class inherited by Image, Volume, and Projection since they share a similar purpose.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 ~Filter()

```
virtual Filter::~Filter ( )  [virtual]
```

The documentation for this class was generated from the following file:

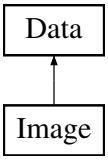
- filter/Filter.h

4.3 Image Class Reference

Class to represent 2D images.

`#include <Image.h>`

Inheritance diagram for Image:



Public Member Functions

- Image ()
- Image (char *file_name)
- Image (unsigned char *data, int w, int h, int c, char *name)
- Image (Image &img)
- Image (Image &&img)
- virtual ~Image ()
- int get_width ()
- int get_height ()
- int get_channels ()

Public Member Functions inherited from Data

- Data ()
- Data (char *file_name)

  Constructor for Data used for single images. is_dir will default to False.
- Data (char *file_name, bool is_dir)

  Constructor for a new data object allowing the specification of is_dir. This is used for volume data, since an entire directory of images is read.
- Data (unsigned char *data, int w, int h, int c, char *name)

  Constructor for Data allowing the specification of a data array along with the dimensions of a single image. It is assumed to be a single image and is_dir will default to False.
- Data (unsigned char *data, int w, int h, int c, int num_files, char *name)

  Constructor for Data allowing the specification of a data array of many constituent images of a volume. It is assumed to be a volume and the maximum z dimension is num_files, the number of images.
- Data (Data &data)
- Data (Data &&data)
- virtual ~Data ()

  Destructor for Data. Ensures that allocated memory is freed.
- void read_dir ()

  Reads all images from a directory and stores it in the object's data variable representing a 3D volume.
- void read_file ()

  Reads the image file and stores it in data variable.
- void read_file (std::filesystem::path file, int channels)

  Reads the image file and stores it in data variable.
- void write_file ()
- void write_file (std::string sub_dir)
- int get_size () const
- void set_input_file (char *file_name)

  Sets the input file of type filesystem::path to the file at file_name.
- void update (unsigned char *data, int channels, int size)

  Updates the object.

Additional Inherited Members

Static Public Member Functions inherited from Data

- static void write_file (unsigned char *write_data, int width, int height, int channels, std::string sub_dir, std::string file_name)

Public Attributes inherited from Data

- char * file_name
- std::filesystem::path input_file
- unsigned char * data
- int w
- int h
- int c
- int size
- int num_files
- bool file_read = false
- bool is_dir = false

4.3.1 Detailed Description

Class to represent 2D images.

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.3.2 Constructor & Destructor Documentation

4.3.2.1 Image() [1/5]

```
Image::Image ( )
```

4.3.2.2 Image() [2/5]

```
Image::Image (
            char * file_name )
```

4.3.2.3 Image() [3/5]

```
Image::Image (
            unsigned char * data,
            int w,
            int h,
            int c,
            char * name )
```

4.3.2.4 Image() [4/5]

```
Image::Image (
            Image & img )
```

4.3.2.5 Image() [5/5]

```
Image::Image (
            Image && img )
```

4.3.2.6 ~Image()

```
Image::~Image ( )  [virtual]
```

4.3.3 Member Function Documentation

4.3.3.1 get_channels()

```
int Image::get_channels ( )
```

4.3.3.2 get_height()

```
int Image::get_height ( )
```

4.3.3.3 get_width()

```
int Image::get_width ( )
```

The documentation for this class was generated from the following files:

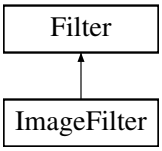
- image/Image.h
- image/Image.cpp

4.4 ImageFilter Class Reference

ImageFilter class which encapsulates filtering methods for 2D images.

```
#include <ImageFilter.h>
```

Inheritance diagram for ImageFilter:



Public Member Functions

- ~ImageFilter () override=default

Public Member Functions inherited from Filter

- virtual ~Filter ()

Static Public Member Functions

- static void adjust_brightness (Image &image, int brightness)

  Adjust the brightness of an image.
- static void to_grayscale (Image &image)

  Convert an image to grayscale.
- static void color_balance (Image &image)

  Apply color balance to an image.
- static void edge_detection (Image &image, std::string filter, int threshold)

  Apply the Sobel filter to an image.
- static void GaussianFilter (Image &image, int kernelSize, double sigma)

  Add Gaussian filter to the image and replace the original image.
- static void median_filter (Image &img, int kernel_size)
- static void histogram_equalization (Image &img)

  histogram equalising an image
- static void BoxFilter (Image &image, int kernelSize)

  Add Box blur filter to the image and replace the original image.

4.4.1 Detailed Description

ImageFilter class which encapsulates filtering methods for 2D images.

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.4.2 Constructor & Destructor Documentation

4.4.2.1 ~ImageFilter()

```
ImageFilter::~ImageFilter ( )  [override], [default]
```

4.4.3 Member Function Documentation

4.4.3.1 adjust_brightness()

```
void ImageFilter::adjust_brightness (
            Image & image,
            int brightness )  [static]
```

Adjust the brightness of an image.

Parameters

in	*image*	
in	*brightness*	

4.4.3.2 BoxFilter()

```
void ImageFilter::BoxFilter (
            Image & image,
            int kernelSize = 5 )  [static]
```

Add Box blur filter to the image and replace the original image.

Parameters

in	*image*	
in	*size*	of the kernal

4.4.3.3 color_balance()

```
void ImageFilter::color_balance (
            Image & image )  [static]
```

Apply color balance to an image.

Parameters

in	*image*	

4.4.3.4 edge_detection()

```
void ImageFilter::edge_detection (
            Image & image,
            std::string filter,
            int threshold )  [static]
```

Apply the Sobel filter to an image.

Parameters

in	*image*	
in	*filter*	Must be one of Sobel, Prewitt, and Scharr
in	*threshold*	

4.4.3.5 GaussianFilter()

```
void ImageFilter::GaussianFilter (
            Image & image,
            int kernelSize = 5,
            double sigma = 1.0 )  [static]
```

Add Gaussian filter to the image and replace the original image.

Parameters

in	*image*	
in	*size*	of the kernal
in	*standard*	deviation for Gaussian function

4.4.3.6 histogram_equalization()

```
void ImageFilter::histogram_equalization (
            Image & img )  [static]
```

histogram equalising an image

Parameters

in	*img*	

4.4.3.7 median_filter()

```
void ImageFilter::median_filter (
            Image & img,
            int kernel_size )  [static]
```

Applies a median filter to an image using the quick-select algorithm.

Parameters

img	The input image.
kernel_size	The size of the median filter kernel.

4.4.3.8 to_grayscale()

```
void ImageFilter::to_grayscale (
            Image & image )  [static]
```

Convert an image to grayscale.

Parameters

in	*image*	

The documentation for this class was generated from the following files:

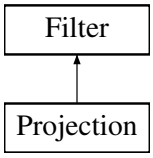
- filter/ImageFilter.h
- filter/ImageFilter.cpp

4.5 Projection Class Reference

Projection class which encapsulates projection methods for 3D volumes.

`#include <Projection.h>`

Inheritance diagram for Projection:



Public Member Functions

- ~Projection () override=default

Public Member Functions inherited from Filter

- virtual ~Filter ()

Static Public Member Functions

- static void max_intensity_projection (Volume &vol, Image &image, bool specify=false, int min_z=0, int max_z=0)

  making maximum intensity projection on a volume and return 2D image
- static void min_intensity_projection (Volume &vol, Image &image, bool specify=false, int min_z=0, int max_z=0)

  making maximum intensity projection on a volume and return 2D image
- static void mean_avg_intensity_projection (Volume &vol, Image &image, bool specify=false, int min_z=0, int max_z=0)

  making maximum intensity projection on a volume and return 2D image
- static void median_avg_intensity_projection (Volume &vol, Image &image, bool specify=false, int min_z=0, int max_z=0)

  making maximum intensity projection on a volume and return 2D image
- static int locating (int i, int j, int k, int w, int l)

4.5.1 Detailed Description

Projection class which encapsulates projection methods for 3D volumes.

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.5.2 Constructor & Destructor Documentation

4.5.2.1 ~Projection()

`Projection::~Projection ( ) [override], [default]`

4.5.3 Member Function Documentation

4.5.3.1 locating()

```
int Projection::locating (
            int i,
            int j,
            int k,
            int w,
            int l )  [static]
```

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.5.3.2 max_intensity_projection()

```
void Projection::max_intensity_projection (
            Volume & vol,
            Image & image,
            bool specify = false,
            int min_z = 0,
            int max_z = 0 )  [static]
```

making maximum intensity projection on a volume and return 2D image

Parameters

in	*vol*	object of volume
in	*image*	object of output image
in	*specify*	buttom allowing user to specify projecting from slab
in	*head*	position of slab
in	*tail*	position of slab

Returns

no return value

4.5.3.3 mean_avg_intensity_projection()

```
void Projection::mean_avg_intensity_projection (
            Volume & vol,
            Image & image,
            bool specify = false,
            int min_z = 0,
            int max_z = 0 )  [static]
```

making maximum intensity projection on a volume and return 2D image

Parameters

in	*vol*	object of volume
in	*image*	object of output image
in	*specify*	buttom allowing user to specify projecting from slab
in	*head*	position of slab
in	*tail*	position of slab

Returns

no return value

4.5.3.4 median_avg_intensity_projection()

```
void Projection::median_avg_intensity_projection (
            Volume & vol,
            Image & image,
            bool specify = false,
            int min_z = 0,
            int max_z = 0 )  [static]
```

making maximum intensity projection on a volume and return 2D image

Parameters

in	*vol*	object of volume
in	*image*	object of output image
in	*specify*	buttom allowing user to specify projecting from slab
in	*head*	position of slab
in	*tail*	position of slab

Returns

no return value

4.5.3.5 min_intensity_projection()

```
void Projection::min_intensity_projection (
            Volume & vol,
            Image & image,
            bool specify = false,
            int min_z = 0,
            int max_z = 0 )  [static]
```

making maximum intensity projection on a volume and return 2D image

Parameters

in	*vol*	object of volume
in	*image*	object of output image
in	*specify*	buttom allowing user to specify projecting from slab
in	*head*	position of slab
in	*tail*	position of slab

Returns

no return value

The documentation for this class was generated from the following files:

- projection/Projection.h
- projection/Projection.cpp

4.6 Slice Class Reference

Slice class which encapsulates slice methods for 3D volumes.

`#include <Slice.h>`

Static Public Member Functions

- static void slice_xz (Volume &vol, Image &output_image, int y)

  Slice a volume along the y-axis and writes the resulting image to the Output directory.
- static void slice_yz (Volume &vol, Image &output_image, int x)

  Slice a volume along the x-axis and writes the resulting image to the Output directory.

4.6.1 Detailed Description

Slice class which encapsulates slice methods for 3D volumes.

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.6.2 Member Function Documentation

4.6.2.1 slice_xz()

```
void Slice::slice_xz (
            Volume & vol,
            Image & output_image,
            int y )  [static]
```

Slice a volume along the y-axis and writes the resulting image to the Output directory.

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

Parameters

vol	the volume to be sliced
output_image	this Image object will be set to represent the slice
y	the y coordinate of the slice

4.6.2.2 slice_yz()

```
void Slice::slice_yz (
            Volume & vol,
            Image & output_image,
            int x )  [static]
```

Slice a volume along the x-axis and writes the resulting image to the Output directory.

Parameters

vol	the volume to be sliced
output_image	this Image object will be set to represent the slice
x	the x coordinate of the slice

The documentation for this class was generated from the following files:

- slice/Slice.h
- slice/Slice.cpp

4.7 stbi_io_callbacks Struct Reference

```
#include <stb_image.h>
```

Public Attributes

- int(∗ read)(void ∗user, char ∗data, int size)
- void(∗ skip)(void ∗user, int n)
- int(∗ eof)(void ∗user)

4.7.1 Member Data Documentation

4.7.1.1 eof

```
int(* stbi_io_callbacks::eof) (void *user)
```

4.7.1.2 read

```
int(* stbi_io_callbacks::read) (void *user, char *data, int size)
```

4.7.1.3 skip

```
void(* stbi_io_callbacks::skip) (void *user, int n)
```

The documentation for this struct was generated from the following file:

- stb/stb_image.h

4.8 Unittest Class Reference

```
#include <Unittest.h>
```

Static Public Member Functions

- static std::tuple< int, int > test_adjust_brightness ()
  test the adjust brightness member function
- static std::tuple< int, int > test_to_grayscale ()
  test the to_gray member function
- static std::tuple< int, int > test_color_balance ()
  test the color balance member function
- static std::tuple< int, int > test_edge_detection ()
  test the Sobel, Prewitt, Scharr edge detection member function
- static std::tuple< int, int > test_GaussianFilter ()
  test the gaussianFilter member function
- static std::tuple< int, int > test_median_filter ()
  test the median_filter member function
- static std::tuple< int, int > test_histogram_equalization ()
  test the histogram equalization member function
- static std::tuple< int, int > test_GaussianFilter3d (int kernelSize, double sigma)
- static std::tuple< int, int > test_3D_median ()
  test the 3D_median_filter member function
- static std::tuple< int, int > test_3D_GaussianFilter ()
  test the 3D_GaussianFilter member function
- static void test_MIP (bool specify=false, int min_z=0, int max_z=0)
  testing maximum intensity projection
- static void test_MinIP (bool specify=false, int min_z=0, int max_z=0)
  testing minimum intensity projection
- static void test_mean_AIP (bool specify=false, int min_z=0, int max_z=0)
  testing mean average intensity projection
- static void test_median_AIP (bool specify=false, int min_z=0, int max_z=0)
  testing median average intensity projection
- static void run_all ()
  runs all unit tests and prints out the total passed and failed.

4.8.1 Detailed Description

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.8.2 Member Function Documentation

4.8.2.1 run_all()

```
void Unittest::run_all ( )  [static]
```

runs all unit tests and prints out the total passed and failed.

4.8.2.2 test_3D_GaussianFilter()

```
std::tuple< int, int > Unittest::test_3D_GaussianFilter ( )  [static]
```

test the 3D_GaussianFilter member function

4.8.2.3 test_3D_median()

```
std::tuple< int, int > Unittest::test_3D_median ( )  [static]
```

test the 3D_median_filter member function

Parameters

in	*kernelSize*	

4.8.2.4 test_adjust_brightness()

```
std::tuple< int, int > Unittest::test_adjust_brightness ( )  [static]
```

test the adjust brightness member function

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.8.2.5 test_color_balance()

```
std::tuple< int, int > Unittest::test_color_balance ( )  [static]
```

test the color balance member function

4.8.2.6 test_edge_detection()

```
std::tuple< int, int > Unittest::test_edge_detection ( )  [static]
```

test the Sobel, Prewitt, Scharr edge detection member function

4.8.2.7 test_GaussianFilter()

```
std::tuple< int, int > Unittest::test_GaussianFilter ( )  [static]
```

test the gaussianFilter member function

4.8.2.8 test_GaussianFilter3d()

```
static std::tuple< int, int > Unittest::test_GaussianFilter3d (
            int kernelSize,
            double sigma )  [static]
```

4.8.2.9 test_histogram_equalization()

```
std::tuple< int, int > Unittest::test_histogram_equalization ( )  [static]
```

test the histogram equalization member function

4.8.2.10 test_mean_AIP()

```
void Unittest::test_mean_AIP (
            bool specify = false,
            int min_z = 0,
            int max_z = 0 )  [static]
```

testing mean average intensity projection

Parameters

in	*head*	position of slab
in	*tail*	position of slab

Returns

no return value

4.8.2.11 test_median_AIP()

```
void Unittest::test_median_AIP (
            bool specify = false,
            int min_z = 0,
            int max_z = 0 )  [static]
```

testing median average intensity projection

Parameters

in	*head*	position of slab
in	*tail*	position of slab

Returns

no return value

4.8.2.12 test_median_filter()

```
std::tuple< int, int > Unittest::test_median_filter ( )  [static]
```

test the median_filter member function

4.8.2.13 test_MinIP()

```
void Unittest::test_MinIP (
            bool specify = false,
            int min_z = 0,
            int max_z = 0 )  [static]
```

testing minimum intensity projection

Parameters

in	*head*	position of slab
in	*tail*	position of slab

Returns

no return value

4.8.2.14 test_MIP()

```
void Unittest::test_MIP (
            bool specify = false,
            int min_z = 0,
            int max_z = 0 )  [static]
```

testing maximum intensity projection

Parameters

in	*head*	position of slab
in	*tail*	position of slab

Returns

no return value

4.8.2.15 test_to_grayscale()

```
std::tuple< int, int > Unittest::test_to_grayscale ( )  [static]
```

test the to_gray member function

The documentation for this class was generated from the following files:

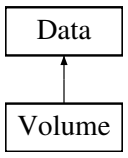
- unittest/Unittest.h
- unittest/Unittest.cpp

4.9 Volume Class Reference

Class to represent 3D volumes.

`#include <Volume.h>`

Inheritance diagram for Volume:



Public Member Functions

- Volume (char *dir_name)
- Volume ()
- Volume (unsigned char *data, int w, int h, int c, int num_files, char *name)
- virtual ~Volume ()
- int get_x ()
- int get_y ()
- int get_z ()
- int get_channels ()

Public Member Functions inherited from Data

- Data ()
- Data (char *file_name)

  Constructor for Data used for single images. is_dir will default to False.
- Data (char *file_name, bool is_dir)

  Constructor for a new data object allowing the specification of is_dir. This is used for volume data, since an entire directory of images is read.
- Data (unsigned char *data, int w, int h, int c, char *name)

  Constructor for Data allowing the specification of a data array along with the dimensions of a single image. It is assumed to be a single image and is_dir will default to False.
- Data (unsigned char *data, int w, int h, int c, int num_files, char *name)

  Constructor for Data allowing the specification of a data array of many constituent images of a volume. It is assumed to be a volume and the maximum z dimension is num_files, the number of images.
- Data (Data &data)
- Data (Data &&data)
- virtual ~Data ()

  Destructor for Data. Ensures that allocated memory is freed.
- void read_dir ()

  Reads all images from a directory and stores it in the object's data variable representing a 3D volume.
- void read_file ()

  Reads the image file and stores it in data variable.
- void read_file (std::filesystem::path file, int channels)

  Reads the image file and stores it in data variable.
- void write_file ()
- void write_file (std::string sub_dir)
- int get_size () const
- void set_input_file (char *file_name)

  Sets the input file of type filesystem::path to the file at file_name.
- void update (unsigned char *data, int channels, int size)

  Updates the object.

Public Attributes

- char * dir_name

Public Attributes inherited from Data

- char * file_name
- std::filesystem::path input_file
- unsigned char * data
- int w
- int h
- int c
- int size
- int num_files
- bool file_read = false
- bool is_dir = false

Additional Inherited Members

Static Public Member Functions inherited from Data

- static void write_file (unsigned char *write_data, int width, int height, int channels, std::string sub_dir, std::string file_name)

4.9.1 Detailed Description

Class to represent 3D volumes.

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.9.2 Constructor & Destructor Documentation

4.9.2.1 Volume() [1/3]

```
Volume::Volume (
            char * dir_name )
```

4.9.2.2 Volume() [2/3]

```
Volume::Volume ( )
```

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.9.2.3 Volume() [3/3]

```
Volume::Volume (
            unsigned char * data,
            int w,
            int h,
            int c,
            int num_files,
            char * name )
```

4.9.2.4 ~Volume()

```
Volume::~Volume ( )  [virtual]
```

4.9.3 Member Function Documentation

4.9.3.1 get_channels()

```
int Volume::get_channels ( )
```

4.9.3.2 get_x()

```
int Volume::get_x ( )
```

4.9.3.3 get_y()

```
int Volume::get_y ( )
```

4.9.3.4 get_z()

```
int Volume::get_z ( )
```

4.9.4 Member Data Documentation

4.9.4.1 dir_name

```
char* Volume::dir_name
```

The documentation for this class was generated from the following files:

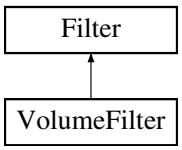
- volume/Volume.h
- volume/Volume.cpp

4.10 VolumeFilter Class Reference

VolumeFilter class which encapsulates filtering methods for 3D volumes.

```
#include <VolumeFilter.h>
```

Inheritance diagram for VolumeFilter:



Public Member Functions

- ~VolumeFilter () override=default

Public Member Functions inherited from Filter

- virtual ~Filter ()

Static Public Member Functions

- static void GaussianFilter3d (Volume &volume, int kernelSize, double sigma)

  Add Gaussian filter to the volume and replace the original volume.
- static void median3D_filter (Volume &img, int kernel_size)

4.10.1 Detailed Description

VolumeFilter class which encapsulates filtering methods for 3D volumes.

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

4.10.2 Constructor & Destructor Documentation

4.10.2.1 ~VolumeFilter()

```
VolumeFilter::~VolumeFilter ( )  [override], [default]
```

4.10.3 Member Function Documentation

4.10.3.1 GaussianFilter3d()

```
void VolumeFilter::GaussianFilter3d (
            Volume & volume,
            int kernelSize = 5,
            double sigma = 1.0 )  [static]
```

Add Gaussian filter to the volume and replace the original volume.

Parameters

in	*volume*	
in	*size*	of the kernal
in	*standard*	deviation for Gaussian function

4.10.3.2 median3D_filter()

```
void VolumeFilter::median3D_filter (
            Volume & vol,
            int kernel_size )  [static]
```

Apply 3D median filter to a given 3D image.

Parameters

vol	The source 3D image.
kernel_size	The size of the kernel to be applied.

The documentation for this class was generated from the following files:

- filter/VolumeFilter.h
- filter/VolumeFilter.cpp

Chapter 5

File Documentation

5.1 data/Data.cpp File Reference

```
#include "Data.h"
#include <string>
#include <iostream>
#include <vector>
#include "../stb/stb_image.h"
#include "../stb/stb_image_write.h"
```

Macros

- #define STB_IMAGE_IMPLEMENTATION
- #define STB_IMAGE_WRITE_IMPLEMENTATION
- #define OUTPUT_DIR "../Output/"

5.1.1 Macro Definition Documentation

5.1.1.1 OUTPUT_DIR

```
#define OUTPUT_DIR "../Output/"
```

5.1.1.2 STB_IMAGE_IMPLEMENTATION

```
#define STB_IMAGE_IMPLEMENTATION
```

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

5.1.1.3 STB_IMAGE_WRITE_IMPLEMENTATION

```
#define STB_IMAGE_WRITE_IMPLEMENTATION
```

5.2 data/Data.h File Reference

```
#include <iostream>
#include <filesystem>
```

Classes

- class Data

5.3 Data.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009
00010 #include <iostream>
00011 #include <filesystem>
00012
00013 // template <class T>
00014 class Data
00015 {
00016 public:
00017     // default constructor - should we have it?
00018     Data();
00019     // constructor
00020     Data(char* file_name);
00021     Data(char* file_name, bool is_dir);
00022     Data(unsigned char* data, int w, int h, int c, char* name);
00023     Data(unsigned char* data, int w, int h, int c, int num_files, char* name);
00024     Data(Data &data);
00025     Data(Data &&data);
00026
00027     // destructor
00028     virtual ~Data();
00029
00030     // Make sure to handle error if stbi_load fails
00031     void read_dir();
00032     void read_file();
00033     void read_file(std::filesystem::path file, int channels);
00034
00035     void write_file();
00036     void write_file(std::string sub_dir);
00037     static void write_file(unsigned char* write_data, int width, int height, int channels, std::string
      sub_dir, std::string file_name);
00038
00039     int get_size() const;
00040
00041     void set_input_file(char* file_name);
00042
00043     void update(unsigned char* data, int channels, int size);
00044
00045     char* file_name;
00046     std::filesystem::path input_file;
00047     unsigned char* data;
00048     int w, h, c;
00049     int size;
00050     int num_files;
00051
00052     bool file_read = false;
00053     bool is_dir = false;
00054
00055
00056 protected:
00057
00058 private:
00059
00060     void insertionSort(std::vector<std::filesystem::path> &vec);
00061
00062 };
```

5.4 filter/Filter.cpp File Reference

5.5 filter/Filter.h File Reference

Classes

- class Filter

5.6 Filter.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009
00015 class Filter
00016 {
00017 public:
00018     virtual ~Filter();
00019 };
```

5.7 filter/ImageFilter.cpp File Reference

```
#include <cstring>
#include <cstdlib>
#include <cmath>
#include <vector>
#include "ImageFilter.h"
```

5.8 filter/ImageFilter.h File Reference

```
#include "Filter.h"
#include "../image/Image.h"
#include "../data/Data.h"
```

Classes

- class ImageFilter

  ImageFilter class which encapsulates filtering methods for 2D images.

5.9 ImageFilter.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009 #include "Filter.h"
00010 #include "../image/Image.h"
00011 #include "../data/Data.h"
00012
00013
00017 class ImageFilter : public Filter
00018 {
00019 public:
00020
00021     ~ImageFilter() override = default;
00022
00023     static void adjust_brightness(Image& image, int brightness);
00024     static void to_grayscale(Image& image);
00025     static void color_balance(Image& image);
00026     static void edge_detection(Image& image, std::string filter, int threshold);
00027     static void GaussianFilter(Image& image, int kernelSize, double sigma);
00028     static void median_filter(Image& img, int kernel_size);
00029     static void histogram_equalization(Image& img);
00030     static void BoxFilter(Image& image, int kernelSize);
00031
00032 protected:
00033
00034 private:
00035     static unsigned char get_increase(Image& image, int channels, float percent_change);
00036     static std::vector<float> generateKernel(float kernelSize, double sigma, std::string type);
00037     static void replicate_padding(Image& src, unsigned char* padded_img, int padding);
00038     static void convolution(Image& image, unsigned char* outputImageData, int halfSize,
      std::vector<float> kernel);
00039     static unsigned char partition(unsigned char* nums, int left, int right);
00040     static unsigned char quick_select(unsigned char* nums, int left, int right, int k);
00041 };
```

5.10 filter/VolumeFilter.cpp File Reference

```
#include <cstring>
#include <cstdlib>
#include <cmath>
#include <vector>
#include "../volume/Volume.h"
#include "VolumeFilter.h"
```

5.11 filter/VolumeFilter.h File Reference

```
#include "Filter.h"
#include "../volume/Volume.h"
#include "../data/Data.h"
```

Classes

- class VolumeFilter

  VolumeFilter class which encapsulates filtering methods for 3D volumes.

5.12 VolumeFilter.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009
00010 #include "Filter.h"
00011 #include "../volume/Volume.h"
00012 #include "../data/Data.h"
00013
00017 class VolumeFilter : public Filter
00018 {
00019 public:
00020
00021     // destructor
00022     ~VolumeFilter() override = default;
00023
00024     static void GaussianFilter3d(Volume &volume,int kernelSize,double sigma);
00025     static void median3D_filter(Volume& img, int kernel_size);
00026
00027 protected:
00028
00029 private:
00030     static std::vector<double> generate_1d_gaussian_kernel(int kernelSize, double sigma);
00031     static void replicate3D_padding(Volume& img, unsigned char* padded_img, int padding);
00032     static unsigned char partition(unsigned char* nums, int left, int right);
00033     static unsigned char quick_select(unsigned char* nums, int left, int right, int k);
00034
00035 };
```

5.13 image/Image.cpp File Reference

```
#include "Image.h"
#include <string>
```

Enumerations

- enum filter { brightness , grayscale }

5.13.1 Enumeration Type Documentation

5.13.1.1 filter

```
enum filter
```

Group name: Tarjan Authors: Eric Brine, Ligen Cai, Yujin Choi, Jinwei Hu, Yu Yan, Hao Zhou Github usernames: edsml-tc2122, acse-yc2122, acse-ericbrine, edsml-yy2622, edsml-hz122, ACSE-jh4322

Enumerator

brightness	
grayscale	

5.14 image/Image.h File Reference

```
#include "../data/Data.h"
```

Classes

- class Image

  Class to represent 2D images.

5.15 Image.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009
00010 #include "../data/Data.h"
00011
00015 class Image : public Data
00016 {
00017 public:
00018     // Constructors
00019     Image();
00020     Image(char* file_name);
00021     Image(unsigned char* data, int w, int h, int c, char* name);
00022     Image(Image &img);
00023     Image(Image &&img);
00024
00025     // destructor
00026     virtual ~Image();
00027
00028     int get_width();
00029     int get_height();
00030     int get_channels();
00031
00032 protected:
00033
00034 private:
00035
00036
00037 };
```

5.16 main.cpp File Reference

```
#include "data/Data.h"
#include "image/Image.h"
#include "volume/Volume.h"
#include "filter/ImageFilter.h"
#include "filter/VolumeFilter.h"
#include "projection/Projection.h"
#include "slice/Slice.h"
#include "unittest/Unittest.h"
#include <string>
#include <sstream>
#include <stdexcept>
```

Functions

- bool is_int (const char *str)
- int main (int argc, const char *argv[])

5.16.1 Function Documentation

5.16.1.1 is_int()

```
bool is_int (
            const char * str )
```

5.16.1.2 main()

```
int main (
            int argc,
            const char * argv[] )
```

5.17 minimal.cpp File Reference

```
#include <iostream>
#include <string>
#include "./stb/stb_image.h"
#include "./stb/stb_image_write.h"
```

Macros

- #define STB_IMAGE_IMPLEMENTATION
- #define STB_IMAGE_WRITE_IMPLEMENTATION

Functions

- int main ()

5.17.1 Macro Definition Documentation

5.17.1.1 STB_IMAGE_IMPLEMENTATION

```
#define STB_IMAGE_IMPLEMENTATION
```

5.17.1.2 STB_IMAGE_WRITE_IMPLEMENTATION

```
#define STB_IMAGE_WRITE_IMPLEMENTATION
```

5.17.2 Function Documentation

5.17.2.1 main()

```
int main ( )
```

5.18 projection/Projection.cpp File Reference

```
#include <cstring>
#include <cstdlib>
#include <algorithm>
#include <cmath>
#include "Projection.h"
#include "../data/Data.h"
```

5.19 projection/Projection.h File Reference

```
#include "../filter/Filter.h"
#include "../volume/Volume.h"
#include "../data/Data.h"
```

Classes

- class Projection

  Projection class which encapsulates projection methods for 3D volumes.

5.20 Projection.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009
00010 #include "../filter/Filter.h"
00011 #include "../volume/Volume.h"
00012 #include "../data/Data.h"
00013
00017 class Projection : public Filter
00018 {
00019 public:
00020
00021     // destructor
00022     ~Projection() override = default;
00023
00024     static void max_intensity_projection(Volume& vol, Image& image, bool specify= false, int min_z= 0,
      int max_z= 0);
00025     static void min_intensity_projection(Volume& vol, Image& image, bool specify= false, int min_z= 0,
      int max_z= 0);
00026     static void mean_avg_intensity_projection(Volume& vol, Image& image, bool specify = false, int
      min_z= 0, int max_z= 0);
00027     static void median_avg_intensity_projection(Volume& vol, Image& image, bool specify= false, int
      min_z= 0, int max_z= 0);
00028     static int locating(int i, int j, int k, int w, int l);
00029 protected:
00030
00031 private:
00032
00033 };
```

5.21 slice/Slice.cpp File Reference

```
#include "Slice.h"
```

5.22 slice/Slice.h File Reference

```
#include <string>
#include "../volume/Volume.h"
#include "../data/Data.h"
#include "../image/Image.h"
```

Classes

- class Slice

  Slice class which encapsulates slice methods for 3D volumes.

5.23 Slice.h

Go to the documentation of this file.

```
00001
00007 #pragma once
00008
00009 #include <string>
00010
00011 #include "../volume/Volume.h"
```

```
00012 #include "../data/Data.h"
00013 #include "../image/Image.h"
00014
00018 class Slice
00019 {
00020 public:
00021
00022     static void slice_xz(Volume& vol, Image& output_image, int y);
00023     static void slice_yz(Volume& vol, Image& output_image, int x);
00024
00025 protected:
00026
00027 private:
00028
00029 };
```

5.24 stb/stb_image.h File Reference

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

Classes

- struct stbi_io_callbacks

Macros

- #define STBI_VERSION 1
- #define STBIDEF extern

Typedefs

- typedef unsigned char stbi_uc
- typedef unsigned short stbi_us

Enumerations

- enum {
  STBI_default = 0 , STBI_grey = 1 , STBI_grey_alpha = 2 , STBI_rgb = 3 ,
  STBI_rgb_alpha = 4 }

Functions

- STBIDEF stbi_uc * stbi_load_from_memory (stbi_uc const *buffer, int len, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF stbi_uc * stbi_load_from_callbacks (stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF stbi_uc * stbi_load (char const *filename, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF stbi_uc * stbi_load_from_file (FILE *f, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF stbi_uc * stbi_load_gif_from_memory (stbi_uc const *buffer, int len, int **delays, int *x, int *y, int *z, int *comp, int req_comp)
- STBIDEF stbi_us * stbi_load_16_from_memory (stbi_uc const *buffer, int len, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF stbi_us * stbi_load_16_from_callbacks (stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF stbi_us * stbi_load_16 (char const *filename, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF stbi_us * stbi_load_from_file_16 (FILE *f, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF float * stbi_loadf_from_memory (stbi_uc const *buffer, int len, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF float * stbi_loadf_from_callbacks (stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF float * stbi_loadf (char const *filename, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF float * stbi_loadf_from_file (FILE *f, int *x, int *y, int *channels_in_file, int desired_channels)
- STBIDEF void stbi_hdr_to_ldr_gamma (float gamma)
- STBIDEF void stbi_hdr_to_ldr_scale (float scale)
- STBIDEF void stbi_ldr_to_hdr_gamma (float gamma)
- STBIDEF void stbi_ldr_to_hdr_scale (float scale)
- STBIDEF int stbi_is_hdr_from_callbacks (stbi_io_callbacks const *clbk, void *user)
- STBIDEF int stbi_is_hdr_from_memory (stbi_uc const *buffer, int len)
- STBIDEF int stbi_is_hdr (char const *filename)
- STBIDEF int stbi_is_hdr_from_file (FILE *f)
- STBIDEF const char * stbi_failure_reason (void)
- STBIDEF void stbi_image_free (void *retval_from_stbi_load)
- STBIDEF int stbi_info_from_memory (stbi_uc const *buffer, int len, int *x, int *y, int *comp)
- STBIDEF int stbi_info_from_callbacks (stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *comp)
- STBIDEF int stbi_is_16_bit_from_memory (stbi_uc const *buffer, int len)
- STBIDEF int stbi_is_16_bit_from_callbacks (stbi_io_callbacks const *clbk, void *user)
- STBIDEF int stbi_info (char const *filename, int *x, int *y, int *comp)
- STBIDEF int stbi_info_from_file (FILE *f, int *x, int *y, int *comp)
- STBIDEF int stbi_is_16_bit (char const *filename)
- STBIDEF int stbi_is_16_bit_from_file (FILE *f)
- STBIDEF void stbi_set_unpremultiply_on_load (int flag_true_if_should_unpremultiply)
- STBIDEF void stbi_convert_iphone_png_to_rgb (int flag_true_if_should_convert)
- STBIDEF void stbi_set_flip_vertically_on_load (int flag_true_if_should_flip)
- STBIDEF void stbi_set_unpremultiply_on_load_thread (int flag_true_if_should_unpremultiply)
- STBIDEF void stbi_convert_iphone_png_to_rgb_thread (int flag_true_if_should_convert)
- STBIDEF void stbi_set_flip_vertically_on_load_thread (int flag_true_if_should_flip)
- STBIDEF char * stbi_zlib_decode_malloc_guesssize (const char *buffer, int len, int initial_size, int *outlen)
- STBIDEF char * stbi_zlib_decode_malloc_guesssize_headerflag (const char *buffer, int len, int initial_size, int *outlen, int parse_header)
- STBIDEF char * stbi_zlib_decode_malloc (const char *buffer, int len, int *outlen)
- STBIDEF int stbi_zlib_decode_buffer (char *obuffer, int olen, const char *ibuffer, int ilen)
- STBIDEF char * stbi_zlib_decode_noheader_malloc (const char *buffer, int len, int *outlen)
- STBIDEF int stbi_zlib_decode_noheader_buffer (char *obuffer, int olen, const char *ibuffer, int ilen)

5.24.1 Macro Definition Documentation

5.24.1.1 STBI_VERSION

```
#define STBI_VERSION 1
```

5.24.1.2 STBIDEF

```
#define STBIDEF extern
```

5.24.2 Typedef Documentation

5.24.2.1 stbi_uc

```
typedef unsigned char stbi_uc
```

5.24.2.2 stbi_us

```
typedef unsigned short stbi_us
```

5.24.3 Enumeration Type Documentation

5.24.3.1 anonymous enum

```
anonymous enum
```

Enumerator

STBI_default	
STBI_grey	
STBI_grey_alpha	
STBI_rgb	
STBI_rgb_alpha	

5.24.4 Function Documentation

5.24.4.1 stbi_convert_iphone_png_to_rgb()

```
STBIDEF void stbi_convert_iphone_png_to_rgb (
            int flag_true_if_should_convert )
```

5.24.4.2 stbi_convert_iphone_png_to_rgb_thread()

```
STBIDEF void stbi_convert_iphone_png_to_rgb_thread (
            int flag_true_if_should_convert )
```

5.24.4.3 stbi_failure_reason()

```
STBIDEF const char * stbi_failure_reason (
            void  )
```

5.24.4.4 stbi_hdr_to_ldr_gamma()

```
STBIDEF void stbi_hdr_to_ldr_gamma (
            float gamma )
```

5.24.4.5 stbi_hdr_to_ldr_scale()

```
STBIDEF void stbi_hdr_to_ldr_scale (
            float scale )
```

5.24.4.6 stbi_image_free()

```
STBIDEF void stbi_image_free (
            void * retval_from_stbi_load )
```

5.24.4.7 stbi_info()

```
STBIDEF int stbi_info (
             char const * filename,
             int * x,
             int * y,
             int * comp )
```

5.24.4.8 stbi_info_from_callbacks()

```
STBIDEF int stbi_info_from_callbacks (
             stbi_io_callbacks const * clbk,
             void * user,
             int * x,
             int * y,
             int * comp )
```

5.24.4.9 stbi_info_from_file()

```
STBIDEF int stbi_info_from_file (
             FILE * f,
             int * x,
             int * y,
             int * comp )
```

5.24.4.10 stbi_info_from_memory()

```
STBIDEF int stbi_info_from_memory (
             stbi_uc const * buffer,
             int len,
             int * x,
             int * y,
             int * comp )
```

5.24.4.11 stbi_is_16_bit()

```
STBIDEF int stbi_is_16_bit (
             char const * filename )
```

5.24.4.12 stbi_is_16_bit_from_callbacks()

```
STBIDEF int stbi_is_16_bit_from_callbacks (
            stbi_io_callbacks const * clbk,
            void * user )
```

5.24.4.13 stbi_is_16_bit_from_file()

```
STBIDEF int stbi_is_16_bit_from_file (
            FILE * f )
```

5.24.4.14 stbi_is_16_bit_from_memory()

```
STBIDEF int stbi_is_16_bit_from_memory (
            stbi_uc const * buffer,
            int len )
```

5.24.4.15 stbi_is_hdr()

```
STBIDEF int stbi_is_hdr (
            char const * filename )
```

5.24.4.16 stbi_is_hdr_from_callbacks()

```
STBIDEF int stbi_is_hdr_from_callbacks (
            stbi_io_callbacks const * clbk,
            void * user )
```

5.24.4.17 stbi_is_hdr_from_file()

```
STBIDEF int stbi_is_hdr_from_file (
            FILE * f )
```

5.24.4.18 stbi_is_hdr_from_memory()

```
STBIDEF int stbi_is_hdr_from_memory (
            stbi_uc const * buffer,
            int len )
```

5.24.4.19 stbi_ldr_to_hdr_gamma()

```
STBIDEF void stbi_ldr_to_hdr_gamma (
            float gamma )
```

5.24.4.20 stbi_ldr_to_hdr_scale()

```
STBIDEF void stbi_ldr_to_hdr_scale (
            float scale )
```

5.24.4.21 stbi_load()

```
STBIDEF stbi_uc * stbi_load (
            char const * filename,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.22 stbi_load_16()

```
STBIDEF stbi_us * stbi_load_16 (
            char const * filename,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.23 stbi_load_16_from_callbacks()

```
STBIDEF stbi_us * stbi_load_16_from_callbacks (
            stbi_io_callbacks const * clbk,
            void * user,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.24 stbi_load_16_from_memory()

```
STBIDEF stbi_us * stbi_load_16_from_memory (
            stbi_uc const * buffer,
            int len,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.25 stbi_load_from_callbacks()

```
STBIDEF stbi_uc * stbi_load_from_callbacks (
            stbi_io_callbacks const * clbk,
            void * user,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.26 stbi_load_from_file()

```
STBIDEF stbi_uc * stbi_load_from_file (
            FILE * f,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.27 stbi_load_from_file_16()

```
STBIDEF stbi_us * stbi_load_from_file_16 (
            FILE * f,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.28 stbi_load_from_memory()

```
STBIDEF stbi_uc * stbi_load_from_memory (
            stbi_uc const * buffer,
            int len,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.29 stbi_load_gif_from_memory()

```
STBIDEF stbi_uc * stbi_load_gif_from_memory (
            stbi_uc const * buffer,
            int len,
            int ** delays,
            int * x,
            int * y,
            int * z,
            int * comp,
            int req_comp )
```

5.24.4.30 stbi_loadf()

```
STBIDEF float * stbi_loadf (
            char const * filename,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.31 stbi_loadf_from_callbacks()

```
STBIDEF float * stbi_loadf_from_callbacks (
            stbi_io_callbacks const * clbk,
            void * user,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.32 stbi_loadf_from_file()

```
STBIDEF float * stbi_loadf_from_file (
            FILE * f,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.33 stbi_loadf_from_memory()

```
STBIDEF float * stbi_loadf_from_memory (
            stbi_uc const * buffer,
            int len,
            int * x,
            int * y,
            int * channels_in_file,
            int desired_channels )
```

5.24.4.34 stbi_set_flip_vertically_on_load()

```
STBIDEF void stbi_set_flip_vertically_on_load (
            int flag_true_if_should_flip )
```

5.24.4.35 stbi_set_flip_vertically_on_load_thread()

```
STBIDEF void stbi_set_flip_vertically_on_load_thread (
            int flag_true_if_should_flip )
```

5.24.4.36 stbi_set_unpremultiply_on_load()

```
STBIDEF void stbi_set_unpremultiply_on_load (
            int flag_true_if_should_unpremultiply )
```

5.24.4.37 stbi_set_unpremultiply_on_load_thread()

```
STBIDEF void stbi_set_unpremultiply_on_load_thread (
            int flag_true_if_should_unpremultiply )
```

5.24.4.38 stbi_zlib_decode_buffer()

```
STBIDEF int stbi_zlib_decode_buffer (
            char * obuffer,
            int olen,
            const char * ibuffer,
            int ilen )
```

5.24.4.39 stbi_zlib_decode_malloc()

```
STBIDEF char * stbi_zlib_decode_malloc (
            const char * buffer,
            int len,
            int * outlen )
```

5.24.4.40 stbi_zlib_decode_malloc_guesssize()

```
STBIDEF char * stbi_zlib_decode_malloc_guesssize (
            const char * buffer,
            int len,
            int initial_size,
            int * outlen )
```

5.24.4.41 stbi_zlib_decode_malloc_guesssize_headerflag()

```
STBIDEF char * stbi_zlib_decode_malloc_guesssize_headerflag (
            const char * buffer,
            int len,
            int initial_size,
            int * outlen,
            int parse_header )
```

5.24.4.42 stbi_zlib_decode_noheader_buffer()

```
STBIDEF int stbi_zlib_decode_noheader_buffer (
             char * obuffer,
             int olen,
             const char * ibuffer,
             int ilen )
```

5.24.4.43 stbi_zlib_decode_noheader_malloc()

```
STBIDEF char * stbi_zlib_decode_noheader_malloc (
             const char * buffer,
             int len,
             int * outlen )
```

5.25 stb_image.h

Go to the documentation of this file.

```
00001 /* stb_image - v2.28 - public domain image loader - http://nothings.org/stb
00002                                   no warranty implied; use at your own risk
00003
00004    Do this:
00005       #define STB_IMAGE_IMPLEMENTATION
00006    before you include this file in *one* C or C++ file to create the implementation.
00007
00008    // i.e. it should look like this:
00009    #include ...
00010    #include ...
00011    #include ...
00012    #define STB_IMAGE_IMPLEMENTATION
00013    #include "stb_image.h"
00014
00015    You can #define STBI_ASSERT(x) before the #include to avoid using assert.h.
00016    And #define STBI_MALLOC, STBI_REALLOC, and STBI_FREE to avoid using malloc,realloc,free
00017
00018
00019    QUICK NOTES:
00020       Primarily of interest to game developers and other people who can
00021           avoid problematic images and only need the trivial interface
00022
00023       JPEG baseline & progressive (12 bpc/arithmetic not supported, same as stock IJG lib)
00024       PNG 1/2/4/8/16-bit-per-channel
00025
00026       TGA (not sure what subset, if a subset)
00027       BMP non-1bpp, non-RLE
00028       PSD (composited view only, no extra channels, 8/16 bit-per-channel)
00029
00030       GIF (*comp always reports as 4-channel)
00031       HDR (radiance rgbE format)
00032       PIC (Softimage PIC)
00033       PNM (PPM and PGM binary only)
00034
00035       Animated GIF still needs a proper API, but here's one way to do it:
00036           http://gist.github.com/urraka/685d9a6340b26b830d49
00037
00038       - decode from memory or through FILE (define STBI_NO_STDIO to remove code)
00039       - decode from arbitrary I/O callbacks
00040       - SIMD acceleration on x86/x64 (SSE2) and ARM (NEON)
00041
00042    Full documentation under "DOCUMENTATION" below.
00043
00044
00045 LICENSE
00046
00047   See end of file for license information.
00048
00049 RECENT REVISION HISTORY:
00050
```

```
00051       2.28  (2023-01-29) many error fixes, security errors, just tons of stuff
00052       2.27  (2021-07-11) document stbi_info better, 16-bit PNM support, bug fixes
00053       2.26  (2020-07-13) many minor fixes
00054       2.25  (2020-02-02) fix warnings
00055       2.24  (2020-02-02) fix warnings; thread-local failure_reason and flip_vertically
00056       2.23  (2019-08-11) fix clang static analysis warning
00057       2.22  (2019-03-04) gif fixes, fix warnings
00058       2.21  (2019-02-25) fix typo in comment
00059       2.20  (2019-02-07) support utf8 filenames in Windows; fix warnings and platform ifdefs
00060       2.19  (2018-02-11) fix warning
00061       2.18  (2018-01-30) fix warnings
00062       2.17  (2018-01-29) bugfix, 1-bit BMP, 16-bitness query, fix warnings
00063       2.16  (2017-07-23) all functions have 16-bit variants; optimizations; bugfixes
00064       2.15  (2017-03-18) fix png-1,2,4; all Imagenet JPGs; no runtime SSE detection on GCC
00065       2.14  (2017-03-03) remove deprecated STBI_JPEG_OLD; fixes for Imagenet JPGs
00066       2.13  (2016-12-04) experimental 16-bit API, only for PNG so far; fixes
00067       2.12  (2016-04-02) fix typo in 2.11 PSD fix that caused crashes
00068       2.11  (2016-04-02) 16-bit PNGS; enable SSE2 in non-gcc x64
00069                           RGB-format JPEG; remove white matting in PSD;
00070                           allocate large structures on the stack;
00071                           correct channel count for PNG & BMP
00072       2.10  (2016-01-22) avoid warning introduced in 2.09
00073       2.09  (2016-01-16) 16-bit TGA; comments in PNM files; STBI_REALLOC_SIZED
00074
00075    See end of file for full revision history.
00076
00077
00078  ============================    Contributors    =========================
00079
00080  Image formats                             Extensions, features
00081     Sean Barrett (jpeg, png, bmp)             Jetro Lauha (stbi_info)
00082     Nicolas Schulz (hdr, psd)                 Martin "SpartanJ" Golini (stbi_info)
00083     Jonathan Dummer (tga)                     James "moose2000" Brown (iPhone PNG)
00084     Jean-Marc Lienher (gif)                   Ben "Disch" Wenger (io callbacks)
00085     Tom Seddon (pic)                          Omar Cornut (1/2/4-bit PNG)
00086     Thatcher Ulrich (psd)                     Nicolas Guillemot (vertical flip)
00087     Ken Miller (pgm, ppm)                     Richard Mitton (16-bit PSD)
00088     github:urraka (animated gif)              Junggon Kim (PNM comments)
00089     Christopher Forseth (animated gif)        Daniel Gibson (16-bit TGA)
00090                                               socks-the-fox (16-bit PNG)
00091                                               Jeremy Sawicki (handle all ImageNet JPGs)
00092  Optimizations & bugfixes                     Mikhail Morozov (1-bit BMP)
00093     Fabian "ryg" Giesen                       Anael Seghezzi (is-16-bit query)
00094     Arseny Kapoulkine                         Simon Breuss (16-bit PNM)
00095     John-Mark Allen
00096     Carmelo J Fdez-Aguera
00097
00098  Bug & warning fixes
00099     Marc LeBlanc            David Woo          Guillaume George     Martins Mozeiko
00100     Christpher Lloyd        Jerry Jansson      Joseph Thomson       Blazej Dariusz Roszkowski
00101     Phil Jordan                                Dave Moore           Roy Eltham
00102     Hayaki Saito            Nathan Reed        Won Chun
00103     Luke Graham             Johan Duparc       Nick Verigakis       the Horde3D community
00104     Thomas Ruf              Ronny Chevalier                         github:rlyeh
00105     Janez Zemva             John Bartholomew   Michal Cichon        github:romigrou
00106     Jonathan Blow           Ken Hamada         Tero Hanninen        github:svdijk
00107     Eugene Golushkov        Laurent Gomila     Cort Stratton        github:snagar
00108     Aruelien Pocheville     Sergio Gonzalez    Thibault Reuille     github:Zelex
00109     Cass Everitt            Ryamond Barbiero                        github:grim210
00110     Paul Du Bois            Engin Manap        Aldo Culquicondor    github:sammyhw
00111     Philipp Wiesemann       Dale Weiler        Oriol Ferrer Mesia   github:phprus
00112     Josh Tobin              Neil Bickford      Matthew Gregan       github:poppolopoppo
00113     Julian Raschke          Gregory Mullen     Christian Floisand   github:darealshinji
00114     Baldur Karlsson         Kevin Schmidt      JR Smith             github:Michaelangel007
00115                             Brad Weinberger    Matvey Cherevko      github:mosra
00116     Luca Sas                Alexander Veselov  Zack Middleton       [reserved]
00117     Ryan C. Gordon          [reserved]                              [reserved]
00118                       DO NOT ADD YOUR NAME HERE
00119
00120                       Jacko Dirks
00121
00122   To add your name to the credits, pick a random blank space in the middle and fill it.
00123   80% of merge conflicts on stb PRs are due to people adding their name at the end
00124   of the credits.
00125 */
00126
00127 #ifndef STBI_INCLUDE_STB_IMAGE_H
00128 #define STBI_INCLUDE_STB_IMAGE_H
00129
00130 // DOCUMENTATION
00131 //
00132 // Limitations:
00133 //    - no 12-bit-per-channel JPEG
00134 //    - no JPEGs with arithmetic coding
00135 //    - GIF always returns *comp=4
00136 //
00137 // Basic usage (see HDR discussion below for HDR usage):
```

```
00138 //    int x,y,n;
00139 //    unsigned char *data = stbi_load(filename, &x, &y, &n, 0);
00140 //    // ... process data if not NULL ...
00141 //    // ... x = width, y = height, n = # 8-bit components per pixel ...
00142 //    // ... replace '0' with '1'..'4' to force that many components per pixel
00143 //    // ... but 'n' will always be the number that it would have been if you said 0
00144 //    stbi_image_free(data);
00145 //
00146 // Standard parameters:
00147 //    int *x                 -- outputs image width in pixels
00148 //    int *y                 -- outputs image height in pixels
00149 //    int *channels_in_file  -- outputs # of image components in image file
00150 //    int desired_channels   -- if non-zero, # of image components requested in result
00151 //
00152 // The return value from an image loader is an 'unsigned char *' which points
00153 // to the pixel data, or NULL on an allocation failure or if the image is
00154 // corrupt or invalid. The pixel data consists of *y scanlines of *x pixels,
00155 // with each pixel consisting of N interleaved 8-bit components; the first
00156 // pixel pointed to is top-left-most in the image. There is no padding between
00157 // image scanlines or between pixels, regardless of format. The number of
00158 // components N is 'desired_channels' if desired_channels is non-zero, or
00159 // *channels_in_file otherwise. If desired_channels is non-zero,
00160 // *channels_in_file has the number of components that _would_ have been
00161 // output otherwise. E.g. if you set desired_channels to 4, you will always
00162 // get RGBA output, but you can check *channels_in_file to see if it's trivially
00163 // opaque because e.g. there were only 3 channels in the source image.
00164 //
00165 // An output image with N components has the following components interleaved
00166 // in this order in each pixel:
00167 //
00168 //     N=#comp     components
00169 //       1           grey
00170 //       2           grey, alpha
00171 //       3           red, green, blue
00172 //       4           red, green, blue, alpha
00173 //
00174 // If image loading fails for any reason, the return value will be NULL,
00175 // and *x, *y, *channels_in_file will be unchanged. The function
00176 // stbi_failure_reason() can be queried for an extremely brief, end-user
00177 // unfriendly explanation of why the load failed. Define STBI_NO_FAILURE_STRINGS
00178 // to avoid compiling these strings at all, and STBI_FAILURE_USERMSG to get slightly
00179 // more user-friendly ones.
00180 //
00181 // Paletted PNG, BMP, GIF, and PIC images are automatically depalettized.
00182 //
00183 // To query the width, height and component count of an image without having to
00184 // decode the full file, you can use the stbi_info family of functions:
00185 //
00186 //   int x,y,n,ok;
00187 //   ok = stbi_info(filename, &x, &y, &n);
00188 //   // returns ok=1 and sets x, y, n if image is a supported format,
00189 //   // 0 otherwise.
00190 //
00191 // Note that stb_image pervasively uses ints in its public API for sizes,
00192 // including sizes of memory buffers. This is now part of the API and thus
00193 // hard to change without causing breakage. As a result, the various image
00194 // loaders all have certain limits on image size; these differ somewhat
00195 // by format but generally boil down to either just under 2GB or just under
00196 // 1GB. When the decoded image would be larger than this, stb_image decoding
00197 // will fail.
00198 //
00199 // Additionally, stb_image will reject image files that have any of their
00200 // dimensions set to a larger value than the configurable STBI_MAX_DIMENSIONS,
00201 // which defaults to 2**24 = 16777216 pixels. Due to the above memory limit,
00202 // the only way to have an image with such dimensions load correctly
00203 // is for it to have a rather extreme aspect ratio. Either way, the
00204 // assumption here is that such larger images are likely to be malformed
00205 // or malicious. If you do need to load an image with individual dimensions
00206 // larger than that, and it still fits in the overall size limit, you can
00207 // #define STBI_MAX_DIMENSIONS on your own to be something larger.
00208 //
00209 // ===========================================================================
00210 //
00211 // UNICODE:
00212 //
00213 //   If compiling for Windows and you wish to use Unicode filenames, compile
00214 //   with
00215 //       #define STBI_WINDOWS_UTF8
00216 //   and pass utf8-encoded filenames. Call stbi_convert_wchar_to_utf8 to convert
00217 //   Windows wchar_t filenames to utf8.
00218 //
00219 // ===========================================================================
00220 //
00221 // Philosophy
00222 //
00223 // stb libraries are designed with the following priorities:
00224 //
```

```
00225 //    1. easy to use
00226 //    2. easy to maintain
00227 //    3. good performance
00228 //
00229 // Sometimes I let "good performance" creep up in priority over "easy to maintain",
00230 // and for best performance I may provide less-easy-to-use APIs that give higher
00231 // performance, in addition to the easy-to-use ones. Nevertheless, it's important
00232 // to keep in mind that from the standpoint of you, a client of this library,
00233 // all you care about is #1 and #3, and stb libraries DO NOT emphasize #3 above all.
00234 //
00235 // Some secondary priorities arise directly from the first two, some of which
00236 // provide more explicit reasons why performance can't be emphasized.
00237 //
00238 //    - Portable ("ease of use")
00239 //    - Small source code footprint ("easy to maintain")
00240 //    - No dependencies ("ease of use")
00241 //
00242 // ============================================================================
00243 //
00244 // I/O callbacks
00245 //
00246 // I/O callbacks allow you to read from arbitrary sources, like packaged
00247 // files or some other source. Data read from callbacks are processed
00248 // through a small internal buffer (currently 128 bytes) to try to reduce
00249 // overhead.
00250 //
00251 // The three functions you must define are "read" (reads some bytes of data),
00252 // "skip" (skips some bytes of data), "eof" (reports if the stream is at the end).
00253 //
00254 // ============================================================================
00255 //
00256 // SIMD support
00257 //
00258 // The JPEG decoder will try to automatically use SIMD kernels on x86 when
00259 // supported by the compiler. For ARM Neon support, you must explicitly
00260 // request it.
00261 //
00262 // (The old do-it-yourself SIMD API is no longer supported in the current
00263 // code.)
00264 //
00265 // On x86, SSE2 will automatically be used when available based on a run-time
00266 // test; if not, the generic C versions are used as a fall-back. On ARM targets,
00267 // the typical path is to have separate builds for NEON and non-NEON devices
00268 // (at least this is true for iOS and Android). Therefore, the NEON support is
00269 // toggled by a build flag: define STBI_NEON to get NEON loops.
00270 //
00271 // If for some reason you do not want to use any of SIMD code, or if
00272 // you have issues compiling it, you can disable it entirely by
00273 // defining STBI_NO_SIMD.
00274 //
00275 // ============================================================================
00276 //
00277 // HDR image support   (disable by defining STBI_NO_HDR)
00278 //
00279 // stb_image supports loading HDR images in general, and currently the Radiance
00280 // .HDR file format specifically. You can still load any file through the existing
00281 // interface; if you attempt to load an HDR file, it will be automatically remapped
00282 // to LDR, assuming gamma 2.2 and an arbitrary scale factor defaulting to 1;
00283 // both of these constants can be reconfigured through this interface:
00284 //
00285 //     stbi_hdr_to_ldr_gamma(2.2f);
00286 //     stbi_hdr_to_ldr_scale(1.0f);
00287 //
00288 // (note, do not use _inverse_ constants; stbi_image will invert them
00289 // appropriately).
00290 //
00291 // Additionally, there is a new, parallel interface for loading files as
00292 // (linear) floats to preserve the full dynamic range:
00293 //
00294 //    float *data = stbi_loadf(filename, &x, &y, &n, 0);
00295 //
00296 // If you load LDR images through this interface, those images will
00297 // be promoted to floating point values, run through the inverse of
00298 // constants corresponding to the above:
00299 //
00300 //     stbi_ldr_to_hdr_scale(1.0f);
00301 //     stbi_ldr_to_hdr_gamma(2.2f);
00302 //
00303 // Finally, given a filename (or an open file or memory block--see header
00304 // file for details) containing image data, you can query for the "most
00305 // appropriate" interface to use (that is, whether the image is HDR or
00306 // not), using:
00307 //
00308 //     stbi_is_hdr(char *filename);
00309 //
00310 // ============================================================================
00311 //
```

```
00312 // iPhone PNG support:
00313 //
00314 // We optionally support converting iPhone-formatted PNGs (which store
00315 // premultiplied BGRA) back to RGB, even though they're internally encoded
00316 // differently. To enable this conversion, call
00317 // stbi_convert_iphone_png_to_rgb(1).
00318 //
00319 // Call stbi_set_unpremultiply_on_load(1) as well to force a divide per
00320 // pixel to remove any premultiplied alpha *only* if the image file explicitly
00321 // says there's premultiplied data (currently only happens in iPhone images,
00322 // and only if iPhone convert-to-rgb processing is on).
00323 //
00324 // ============================================================================
00325 //
00326 // ADDITIONAL CONFIGURATION
00327 //
00328 //  - You can suppress implementation of any of the decoders to reduce
00329 //    your code footprint by #defining one or more of the following
00330 //    symbols before creating the implementation.
00331 //
00332 //        STBI_NO_JPEG
00333 //        STBI_NO_PNG
00334 //        STBI_NO_BMP
00335 //        STBI_NO_PSD
00336 //        STBI_NO_TGA
00337 //        STBI_NO_GIF
00338 //        STBI_NO_HDR
00339 //        STBI_NO_PIC
00340 //        STBI_NO_PNM   (.ppm and .pgm)
00341 //
00342 //  - You can request *only* certain decoders and suppress all other ones
00343 //    (this will be more forward-compatible, as addition of new decoders
00344 //    doesn't require you to disable them explicitly):
00345 //
00346 //        STBI_ONLY_JPEG
00347 //        STBI_ONLY_PNG
00348 //        STBI_ONLY_BMP
00349 //        STBI_ONLY_PSD
00350 //        STBI_ONLY_TGA
00351 //        STBI_ONLY_GIF
00352 //        STBI_ONLY_HDR
00353 //        STBI_ONLY_PIC
00354 //        STBI_ONLY_PNM   (.ppm and .pgm)
00355 //
00356 //   - If you use STBI_NO_PNG (or _ONLY_ without PNG), and you still
00357 //     want the zlib decoder to be available, #define STBI_SUPPORT_ZLIB
00358 //
00359 //  - If you define STBI_MAX_DIMENSIONS, stb_image will reject images greater
00360 //    than that size (in either width or height) without further processing.
00361 //    This is to let programs in the wild set an upper bound to prevent
00362 //    denial-of-service attacks on untrusted data, as one could generate a
00363 //    valid image of gigantic dimensions and force stb_image to allocate a
00364 //    huge block of memory and spend disproportionate time decoding it. By
00365 //    default this is set to (1 << 24), which is 16777216, but that's still
00366 //    very big.
00367
00368 #ifndef STBI_NO_STDIO
00369 #include <stdio.h>
00370 #endif // STBI_NO_STDIO
00371
00372 #define STBI_VERSION 1
00373
00374 enum
00375 {
00376    STBI_default = 0, // only used for desired_channels
00377
00378    STBI_grey       = 1,
00379    STBI_grey_alpha = 2,
00380    STBI_rgb        = 3,
00381    STBI_rgb_alpha  = 4
00382 };
00383
00384 #include <stdlib.h>
00385 typedef unsigned char stbi_uc;
00386 typedef unsigned short stbi_us;
00387
00388 #ifdef __cplusplus
00389 extern "C" {
00390 #endif
00391
00392 #ifndef STBIDEF
00393 #ifdef STB_IMAGE_STATIC
00394 #define STBIDEF static
00395 #else
00396 #define STBIDEF extern
00397 #endif
00398 #endif
```

```
00399
00401 //
00402 // PRIMARY API - works on images of any type
00403 //
00404
00405 //
00406 // load image by filename, open file, or memory buffer
00407 //
00408
00409 typedef struct
00410 {
00411    int      (*read)  (void *user,char *data,int size);   // fill 'data' with 'size' bytes.  return
      number of bytes actually read
00412    void     (*skip)  (void *user,int n);                 // skip the next 'n' bytes, or 'unget' the
      last -n bytes if negative
00413    int      (*eof)   (void *user);                       // returns nonzero if we are at end of
      file/data
00414 } stbi_io_callbacks;
00415
00417 //
00418 // 8-bits-per-channel interface
00419 //
00420
00421 STBIDEF stbi_uc *stbi_load_from_memory   (stbi_uc           const *buffer, int len   , int *x, int *y,
      int *channels_in_file, int desired_channels);
00422 STBIDEF stbi_uc *stbi_load_from_callbacks(stbi_io_callbacks const *clbk  , void *user, int *x, int *y,
      int *channels_in_file, int desired_channels);
00423
00424 #ifndef STBI_NO_STDIO
00425 STBIDEF stbi_uc *stbi_load            (char const *filename, int *x, int *y, int *channels_in_file,
      int desired_channels);
00426 STBIDEF stbi_uc *stbi_load_from_file  (FILE *f, int *x, int *y, int *channels_in_file, int
      desired_channels);
00427 // for stbi_load_from_file, file pointer is left pointing immediately after image
00428 #endif
00429
00430 #ifndef STBI_NO_GIF
00431 STBIDEF stbi_uc *stbi_load_gif_from_memory(stbi_uc const *buffer, int len, int **delays, int *x, int
      *y, int *z, int *comp, int req_comp);
00432 #endif
00433
00434 #ifdef STBI_WINDOWS_UTF8
00435 STBIDEF int stbi_convert_wchar_to_utf8(char *buffer, size_t bufferlen, const wchar_t* input);
00436 #endif
00437
00439 //
00440 // 16-bits-per-channel interface
00441 //
00442
00443 STBIDEF stbi_us *stbi_load_16_from_memory   (stbi_uc const *buffer, int len, int *x, int *y, int
      *channels_in_file, int desired_channels);
00444 STBIDEF stbi_us *stbi_load_16_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int
      *y, int *channels_in_file, int desired_channels);
00445
00446 #ifndef STBI_NO_STDIO
00447 STBIDEF stbi_us *stbi_load_16          (char const *filename, int *x, int *y, int *channels_in_file,
      int desired_channels);
00448 STBIDEF stbi_us *stbi_load_from_file_16(FILE *f, int *x, int *y, int *channels_in_file, int
      desired_channels);
00449 #endif
00450
00452 //
00453 // float-per-channel interface
00454 //
00455 #ifndef STBI_NO_LINEAR
00456    STBIDEF float *stbi_loadf_from_memory     (stbi_uc const *buffer, int len, int *x, int *y, int
      *channels_in_file, int desired_channels);
00457    STBIDEF float *stbi_loadf_from_callbacks  (stbi_io_callbacks const *clbk, void *user, int *x, int
      *y,  int *channels_in_file, int desired_channels);
00458
00459    #ifndef STBI_NO_STDIO
00460    STBIDEF float *stbi_loadf            (char const *filename, int *x, int *y, int *channels_in_file,
      int desired_channels);
00461    STBIDEF float *stbi_loadf_from_file  (FILE *f, int *x, int *y, int *channels_in_file, int
      desired_channels);
00462    #endif
00463 #endif
00464
00465 #ifndef STBI_NO_HDR
00466    STBIDEF void   stbi_hdr_to_ldr_gamma(float gamma);
00467    STBIDEF void   stbi_hdr_to_ldr_scale(float scale);
00468 #endif // STBI_NO_HDR
00469
00470 #ifndef STBI_NO_LINEAR
00471    STBIDEF void   stbi_ldr_to_hdr_gamma(float gamma);
00472    STBIDEF void   stbi_ldr_to_hdr_scale(float scale);
00473 #endif // STBI_NO_LINEAR
```

```
00474
00475 // stbi_is_hdr is always defined, but always returns false if STBI_NO_HDR
00476 STBIDEF int    stbi_is_hdr_from_callbacks(stbi_io_callbacks const *clbk, void *user);
00477 STBIDEF int    stbi_is_hdr_from_memory(stbi_uc const *buffer, int len);
00478 #ifndef STBI_NO_STDIO
00479 STBIDEF int      stbi_is_hdr          (char const *filename);
00480 STBIDEF int      stbi_is_hdr_from_file(FILE *f);
00481 #endif // STBI_NO_STDIO
00482
00483
00484 // get a VERY brief reason for failure
00485 // on most compilers (and ALL modern mainstream compilers) this is threadsafe
00486 STBIDEF const char *stbi_failure_reason  (void);
00487
00488 // free the loaded image -- this is just free()
00489 STBIDEF void     stbi_image_free      (void *retval_from_stbi_load);
00490
00491 // get image dimensions & components without fully decoding
00492 STBIDEF int      stbi_info_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp);
00493 STBIDEF int      stbi_info_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y,
      int *comp);
00494 STBIDEF int      stbi_is_16_bit_from_memory(stbi_uc const *buffer, int len);
00495 STBIDEF int      stbi_is_16_bit_from_callbacks(stbi_io_callbacks const *clbk, void *user);
00496
00497 #ifndef STBI_NO_STDIO
00498 STBIDEF int      stbi_info              (char const *filename,     int *x, int *y, int *comp);
00499 STBIDEF int      stbi_info_from_file     (FILE *f,                  int *x, int *y, int *comp);
00500 STBIDEF int      stbi_is_16_bit         (char const *filename);
00501 STBIDEF int      stbi_is_16_bit_from_file(FILE *f);
00502 #endif
00503
00504
00505
00506 // for image formats that explicitly notate that they have premultiplied alpha,
00507 // we just return the colors as stored in the file. set this flag to force
00508 // unpremultiplication. results are undefined if the unpremultiply overflow.
00509 STBIDEF void stbi_set_unpremultiply_on_load(int flag_true_if_should_unpremultiply);
00510
00511 // indicate whether we should process iphone images back to canonical format,
00512 // or just pass them through "as-is"
00513 STBIDEF void stbi_convert_iphone_png_to_rgb(int flag_true_if_should_convert);
00514
00515 // flip the image vertically, so the first pixel in the output array is the bottom left
00516 STBIDEF void stbi_set_flip_vertically_on_load(int flag_true_if_should_flip);
00517
00518 // as above, but only applies to images loaded on the thread that calls the function
00519 // this function is only available if your compiler supports thread-local variables;
00520 // calling it will fail to link if your compiler doesn't
00521 STBIDEF void stbi_set_unpremultiply_on_load_thread(int flag_true_if_should_unpremultiply);
00522 STBIDEF void stbi_convert_iphone_png_to_rgb_thread(int flag_true_if_should_convert);
00523 STBIDEF void stbi_set_flip_vertically_on_load_thread(int flag_true_if_should_flip);
00524
00525 // ZLIB client - used by PNG, available for other purposes
00526
00527 STBIDEF char *stbi_zlib_decode_malloc_guesssize(const char *buffer, int len, int initial_size, int
      *outlen);
00528 STBIDEF char *stbi_zlib_decode_malloc_guesssize_headerflag(const char *buffer, int len, int
      initial_size, int *outlen, int parse_header);
00529 STBIDEF char *stbi_zlib_decode_malloc(const char *buffer, int len, int *outlen);
00530 STBIDEF int   stbi_zlib_decode_buffer(char *obuffer, int olen, const char *ibuffer, int ilen);
00531
00532 STBIDEF char *stbi_zlib_decode_noheader_malloc(const char *buffer, int len, int *outlen);
00533 STBIDEF int   stbi_zlib_decode_noheader_buffer(char *obuffer, int olen, const char *ibuffer, int
      ilen);
00534
00535
00536 #ifdef __cplusplus
00537 }
00538 #endif
00539
00540 //
00541 //
00543 #endif // STBI_INCLUDE_STB_IMAGE_H
00544
00545 #ifdef STB_IMAGE_IMPLEMENTATION
00546
00547 #if defined(STBI_ONLY_JPEG) || defined(STBI_ONLY_PNG) || defined(STBI_ONLY_BMP) \
00548   || defined(STBI_ONLY_TGA) || defined(STBI_ONLY_GIF) || defined(STBI_ONLY_PSD) \
00549   || defined(STBI_ONLY_HDR) || defined(STBI_ONLY_PIC) || defined(STBI_ONLY_PNM) \
00550   || defined(STBI_ONLY_ZLIB)
00551    #ifndef STBI_ONLY_JPEG
00552    #define STBI_NO_JPEG
00553    #endif
00554    #ifndef STBI_ONLY_PNG
00555    #define STBI_NO_PNG
00556    #endif
00557    #ifndef STBI_ONLY_BMP
```

```
00558    #define STBI_NO_BMP
00559    #endif
00560    #ifndef STBI_ONLY_PSD
00561    #define STBI_NO_PSD
00562    #endif
00563    #ifndef STBI_ONLY_TGA
00564    #define STBI_NO_TGA
00565    #endif
00566    #ifndef STBI_ONLY_GIF
00567    #define STBI_NO_GIF
00568    #endif
00569    #ifndef STBI_ONLY_HDR
00570    #define STBI_NO_HDR
00571    #endif
00572    #ifndef STBI_ONLY_PIC
00573    #define STBI_NO_PIC
00574    #endif
00575    #ifndef STBI_ONLY_PNM
00576    #define STBI_NO_PNM
00577    #endif
00578 #endif
00579
00580 #if defined(STBI_NO_PNG) && !defined(STBI_SUPPORT_ZLIB) && !defined(STBI_NO_ZLIB)
00581 #define STBI_NO_ZLIB
00582 #endif
00583
00584
00585 #include <stdarg.h>
00586 #include <stddef.h> // ptrdiff_t on osx
00587 #include <stdlib.h>
00588 #include <string.h>
00589 #include <limits.h>
00590
00591 #if !defined(STBI_NO_LINEAR) || !defined(STBI_NO_HDR)
00592 #include <math.h>  // ldexp, pow
00593 #endif
00594
00595 #ifndef STBI_NO_STDIO
00596 #include <stdio.h>
00597 #endif
00598
00599 #ifndef STBI_ASSERT
00600 #include <assert.h>
00601 #define STBI_ASSERT(x) assert(x)
00602 #endif
00603
00604 #ifdef __cplusplus
00605 #define STBI_EXTERN extern "C"
00606 #else
00607 #define STBI_EXTERN extern
00608 #endif
00609
00610
00611 #ifndef _MSC_VER
00612    #ifdef __cplusplus
00613    #define stbi_inline inline
00614    #else
00615    #define stbi_inline
00616    #endif
00617 #else
00618    #define stbi_inline __forceinline
00619 #endif
00620
00621 #ifndef STBI_NO_THREAD_LOCALS
00622    #if defined(__cplusplus) &&  __cplusplus >= 201103L
00623       #define STBI_THREAD_LOCAL       thread_local
00624    #elif defined(__GNUC__) && __GNUC__ < 5
00625       #define STBI_THREAD_LOCAL       __thread
00626    #elif defined(_MSC_VER)
00627       #define STBI_THREAD_LOCAL       __declspec(thread)
00628    #elif defined (__STDC_VERSION__) && __STDC_VERSION__ >= 201112L && !defined(__STDC_NO_THREADS__)
00629       #define STBI_THREAD_LOCAL       _Thread_local
00630    #endif
00631
00632    #ifndef STBI_THREAD_LOCAL
00633       #if defined(__GNUC__)
00634         #define STBI_THREAD_LOCAL       __thread
00635       #endif
00636    #endif
00637 #endif
00638
00639 #if defined(_MSC_VER) || defined(__SYMBIAN32__)
00640 typedef unsigned short stbi__uint16;
00641 typedef   signed short stbi__int16;
00642 typedef unsigned int   stbi__uint32;
00643 typedef   signed int   stbi__int32;
00644 #else
```

```
00645 #include <stdint.h>
00646 typedef uint16_t stbi__uint16;
00647 typedef int16_t  stbi__int16;
00648 typedef uint32_t stbi__uint32;
00649 typedef int32_t  stbi__int32;
00650 #endif
00651
00652 // should produce compiler error if size is wrong
00653 typedef unsigned char validate_uint32[sizeof(stbi__uint32)==4 ? 1 : -1];
00654
00655 #ifdef _MSC_VER
00656 #define STBI_NOTUSED(v)  (void)(v)
00657 #else
00658 #define STBI_NOTUSED(v)  (void)sizeof(v)
00659 #endif
00660
00661 #ifdef _MSC_VER
00662 #define STBI_HAS_LROTL
00663 #endif
00664
00665 #ifdef STBI_HAS_LROTL
00666    #define stbi_lrot(x,y)  _lrotl(x,y)
00667 #else
00668    #define stbi_lrot(x,y)  (((x) << (y)) | ((x) >> (-(y) & 31)))
00669 #endif
00670
00671 #if defined(STBI_MALLOC) && defined(STBI_FREE) && (defined(STBI_REALLOC) ||
      defined(STBI_REALLOC_SIZED))
00672 // ok
00673 #elif !defined(STBI_MALLOC) && !defined(STBI_FREE) && !defined(STBI_REALLOC) &&
      !defined(STBI_REALLOC_SIZED)
00674 // ok
00675 #else
00676 #error "Must define all or none of STBI_MALLOC, STBI_FREE, and STBI_REALLOC (or STBI_REALLOC_SIZED)."
00677 #endif
00678
00679 #ifndef STBI_MALLOC
00680 #define STBI_MALLOC(sz)           malloc(sz)
00681 #define STBI_REALLOC(p,newsz)     realloc(p,newsz)
00682 #define STBI_FREE(p)              free(p)
00683 #endif
00684
00685 #ifndef STBI_REALLOC_SIZED
00686 #define STBI_REALLOC_SIZED(p,oldsz,newsz) STBI_REALLOC(p,newsz)
00687 #endif
00688
00689 // x86/x64 detection
00690 #if defined(__x86_64__) || defined(_M_X64)
00691 #define STBI__X64_TARGET
00692 #elif defined(__i386) || defined(_M_IX86)
00693 #define STBI__X86_TARGET
00694 #endif
00695
00696 #if defined(__GNUC__) && defined(STBI__X86_TARGET) && !defined(__SSE2__) && !defined(STBI_NO_SIMD)
00697 // gcc doesn't support sse2 intrinsics unless you compile with -msse2,
00698 // which in turn means it gets to use SSE2 everywhere. This is unfortunate,
00699 // but previous attempts to provide the SSE2 functions with runtime
00700 // detection caused numerous issues. The way architecture extensions are
00701 // exposed in GCC/Clang is, sadly, not really suited for one-file libs.
00702 // New behavior: if compiled with -msse2, we use SSE2 without any
00703 // detection; if not, we don't use it at all.
00704 #define STBI_NO_SIMD
00705 #endif
00706
00707 #if defined(__MINGW32__) && defined(STBI__X86_TARGET) && !defined(STBI_MINGW_ENABLE_SSE2) &&
      !defined(STBI_NO_SIMD)
00708 // Note that __MINGW32__ doesn't actually mean 32-bit, so we have to avoid STBI__X64_TARGET
00709 //
00710 // 32-bit MinGW wants ESP to be 16-byte aligned, but this is not in the
00711 // Windows ABI and VC++ as well as Windows DLLs don't maintain that invariant.
00712 // As a result, enabling SSE2 on 32-bit MinGW is dangerous when not
00713 // simultaneously enabling "-mstackrealign".
00714 //
00715 // See https://github.com/nothings/stb/issues/81 for more information.
00716 //
00717 // So default to no SSE2 on 32-bit MinGW. If you've read this far and added
00718 // -mstackrealign to your build settings, feel free to #define STBI_MINGW_ENABLE_SSE2.
00719 #define STBI_NO_SIMD
00720 #endif
00721
00722 #if !defined(STBI_NO_SIMD) && (defined(STBI__X86_TARGET) || defined(STBI__X64_TARGET))
00723 #define STBI_SSE2
00724 #include <emmintrin.h>
00725
00726 #ifdef _MSC_VER
00727
00728 #if _MSC_VER >= 1400  // not VC6
```

```
00729 #include <intrin.h> // __cpuid
00730 static int stbi__cpuid3(void)
00731 {
00732    int info[4];
00733    __cpuid(info,1);
00734    return info[3];
00735 }
00736 #else
00737 static int stbi__cpuid3(void)
00738 {
00739    int res;
00740    __asm {
00741       mov  eax,1
00742       cpuid
00743       mov  res,edx
00744    }
00745    return res;
00746 }
00747 #endif
00748
00749 #define STBI_SIMD_ALIGN(type, name) __declspec(align(16)) type name
00750
00751 #if !defined(STBI_NO_JPEG) && defined(STBI_SSE2)
00752 static int stbi__sse2_available(void)
00753 {
00754    int info3 = stbi__cpuid3();
00755    return ((info3 >> 26) & 1) != 0;
00756 }
00757 #endif
00758
00759 #else // assume GCC-style if not VC++
00760 #define STBI_SIMD_ALIGN(type, name) type name __attribute__((aligned(16)))
00761
00762 #if !defined(STBI_NO_JPEG) && defined(STBI_SSE2)
00763 static int stbi__sse2_available(void)
00764 {
00765    // If we're even attempting to compile this on GCC/Clang, that means
00766    // -msse2 is on, which means the compiler is allowed to use SSE2
00767    // instructions at will, and so are we.
00768    return 1;
00769 }
00770 #endif
00771
00772 #endif
00773 #endif
00774
00775 // ARM NEON
00776 #if defined(STBI_NO_SIMD) && defined(STBI_NEON)
00777 #undef STBI_NEON
00778 #endif
00779
00780 #ifdef STBI_NEON
00781 #include <arm_neon.h>
00782 #ifdef _MSC_VER
00783 #define STBI_SIMD_ALIGN(type, name) __declspec(align(16)) type name
00784 #else
00785 #define STBI_SIMD_ALIGN(type, name) type name __attribute__((aligned(16)))
00786 #endif
00787 #endif
00788
00789 #ifndef STBI_SIMD_ALIGN
00790 #define STBI_SIMD_ALIGN(type, name) type name
00791 #endif
00792
00793 #ifndef STBI_MAX_DIMENSIONS
00794 #define STBI_MAX_DIMENSIONS (1 << 24)
00795 #endif
00796
00798 //
00799 //  stbi__context struct and start_xxx functions
00800
00801 // stbi__context structure is our basic context used by all images, so it
00802 // contains all the IO context, plus some basic image information
00803 typedef struct
00804 {
00805    stbi__uint32 img_x, img_y;
00806    int img_n, img_out_n;
00807
00808    stbi_io_callbacks io;
00809    void *io_user_data;
00810
00811    int read_from_callbacks;
00812    int buflen;
00813    stbi_uc buffer_start[128];
00814    int callback_already_read;
00815
00816    stbi_uc *img_buffer, *img_buffer_end;
```

```
00817    stbi_uc *img_buffer_original, *img_buffer_original_end;
00818 } stbi__context;
00819
00820
00821 static void stbi__refill_buffer(stbi__context *s);
00822
00823 // initialize a memory-decode context
00824 static void stbi__start_mem(stbi__context *s, stbi_uc const *buffer, int len)
00825 {
00826    s->io.read = NULL;
00827    s->read_from_callbacks = 0;
00828    s->callback_already_read = 0;
00829    s->img_buffer = s->img_buffer_original = (stbi_uc *) buffer;
00830    s->img_buffer_end = s->img_buffer_original_end = (stbi_uc *) buffer+len;
00831 }
00832
00833 // initialize a callback-based context
00834 static void stbi__start_callbacks(stbi__context *s, stbi_io_callbacks *c, void *user)
00835 {
00836    s->io = *c;
00837    s->io_user_data = user;
00838    s->buflen = sizeof(s->buffer_start);
00839    s->read_from_callbacks = 1;
00840    s->callback_already_read = 0;
00841    s->img_buffer = s->img_buffer_original = s->buffer_start;
00842    stbi__refill_buffer(s);
00843    s->img_buffer_original_end = s->img_buffer_end;
00844 }
00845
00846 #ifndef STBI_NO_STDIO
00847
00848 static int stbi__stdio_read(void *user, char *data, int size)
00849 {
00850    return (int) fread(data,1,size,(FILE*) user);
00851 }
00852
00853 static void stbi__stdio_skip(void *user, int n)
00854 {
00855    int ch;
00856    fseek((FILE*) user, n, SEEK_CUR);
00857    ch = fgetc((FILE*) user);  /* have to read a byte to reset feof()'s flag */
00858    if (ch != EOF) {
00859       ungetc(ch, (FILE *) user);  /* push byte back onto stream if valid. */
00860    }
00861 }
00862
00863 static int stbi__stdio_eof(void *user)
00864 {
00865    return feof((FILE*) user) || ferror((FILE *) user);
00866 }
00867
00868 static stbi_io_callbacks stbi__stdio_callbacks =
00869 {
00870    stbi__stdio_read,
00871    stbi__stdio_skip,
00872    stbi__stdio_eof,
00873 };
00874
00875 static void stbi__start_file(stbi__context *s, FILE *f)
00876 {
00877    stbi__start_callbacks(s, &stbi__stdio_callbacks, (void *) f);
00878 }
00879
00880 //static void stop_file(stbi__context *s) { }
00881
00882 #endif // !STBI_NO_STDIO
00883
00884 static void stbi__rewind(stbi__context *s)
00885 {
00886    // conceptually rewind SHOULD rewind to the beginning of the stream,
00887    // but we just rewind to the beginning of the initial buffer, because
00888    // we only use it after doing 'test', which only ever looks at at most 92 bytes
00889    s->img_buffer = s->img_buffer_original;
00890    s->img_buffer_end = s->img_buffer_original_end;
00891 }
00892
00893 enum
00894 {
00895    STBI_ORDER_RGB,
00896    STBI_ORDER_BGR
00897 };
00898
00899 typedef struct
00900 {
00901    int bits_per_channel;
00902    int num_channels;
00903    int channel_order;
```

```
00904 } stbi__result_info;
00905
00906 #ifndef STBI_NO_JPEG
00907 static int      stbi__jpeg_test(stbi__context *s);
00908 static void    *stbi__jpeg_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri);
00909 static int      stbi__jpeg_info(stbi__context *s, int *x, int *y, int *comp);
00910 #endif
00911
00912 #ifndef STBI_NO_PNG
00913 static int      stbi__png_test(stbi__context *s);
00914 static void    *stbi__png_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri);
00915 static int      stbi__png_info(stbi__context *s, int *x, int *y, int *comp);
00916 static int      stbi__png_is16(stbi__context *s);
00917 #endif
00918
00919 #ifndef STBI_NO_BMP
00920 static int      stbi__bmp_test(stbi__context *s);
00921 static void    *stbi__bmp_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri);
00922 static int      stbi__bmp_info(stbi__context *s, int *x, int *y, int *comp);
00923 #endif
00924
00925 #ifndef STBI_NO_TGA
00926 static int      stbi__tga_test(stbi__context *s);
00927 static void    *stbi__tga_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri);
00928 static int      stbi__tga_info(stbi__context *s, int *x, int *y, int *comp);
00929 #endif
00930
00931 #ifndef STBI_NO_PSD
00932 static int      stbi__psd_test(stbi__context *s);
00933 static void    *stbi__psd_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri, int bpc);
00934 static int      stbi__psd_info(stbi__context *s, int *x, int *y, int *comp);
00935 static int      stbi__psd_is16(stbi__context *s);
00936 #endif
00937
00938 #ifndef STBI_NO_HDR
00939 static int      stbi__hdr_test(stbi__context *s);
00940 static float   *stbi__hdr_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri);
00941 static int      stbi__hdr_info(stbi__context *s, int *x, int *y, int *comp);
00942 #endif
00943
00944 #ifndef STBI_NO_PIC
00945 static int      stbi__pic_test(stbi__context *s);
00946 static void    *stbi__pic_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri);
00947 static int      stbi__pic_info(stbi__context *s, int *x, int *y, int *comp);
00948 #endif
00949
00950 #ifndef STBI_NO_GIF
00951 static int      stbi__gif_test(stbi__context *s);
00952 static void    *stbi__gif_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri);
00953 static void    *stbi__load_gif_main(stbi__context *s, int **delays, int *x, int *y, int *z, int *comp,
      int req_comp);
00954 static int      stbi__gif_info(stbi__context *s, int *x, int *y, int *comp);
00955 #endif
00956
00957 #ifndef STBI_NO_PNM
00958 static int      stbi__pnm_test(stbi__context *s);
00959 static void    *stbi__pnm_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri);
00960 static int      stbi__pnm_info(stbi__context *s, int *x, int *y, int *comp);
00961 static int      stbi__pnm_is16(stbi__context *s);
00962 #endif
00963
00964 static
00965 #ifdef STBI_THREAD_LOCAL
00966 STBI_THREAD_LOCAL
00967 #endif
00968 const char *stbi__g_failure_reason;
00969
00970 STBIDEF const char *stbi_failure_reason(void)
00971 {
00972    return stbi__g_failure_reason;
00973 }
00974
00975 #ifndef STBI_NO_FAILURE_STRINGS
00976 static int stbi__err(const char *str)
00977 {
00978    stbi__g_failure_reason = str;
00979    return 0;
00980 }
```

```
00981 #endif
00982
00983 static void *stbi__malloc(size_t size)
00984 {
00985     return STBI_MALLOC(size);
00986 }
00987
00988 // stb_image uses ints pervasively, including for offset calculations.
00989 // therefore the largest decoded image size we can support with the
00990 // current code, even on 64-bit targets, is INT_MAX. this is not a
00991 // significant limitation for the intended use case.
00992 //
00993 // we do, however, need to make sure our size calculations don't
00994 // overflow. hence a few helper functions for size calculations that
00995 // multiply integers together, making sure that they're non-negative
00996 // and no overflow occurs.
00997
00998 // return 1 if the sum is valid, 0 on overflow.
00999 // negative terms are considered invalid.
01000 static int stbi__addsizes_valid(int a, int b)
01001 {
01002    if (b < 0) return 0;
01003    // now 0 <= b <= INT_MAX, hence also
01004    // 0 <= INT_MAX - b <= INTMAX.
01005    // And "a + b <= INT_MAX" (which might overflow) is the
01006    // same as a <= INT_MAX - b (no overflow)
01007    return a <= INT_MAX - b;
01008 }
01009
01010 // returns 1 if the product is valid, 0 on overflow.
01011 // negative factors are considered invalid.
01012 static int stbi__mul2sizes_valid(int a, int b)
01013 {
01014    if (a < 0 || b < 0) return 0;
01015    if (b == 0) return 1; // mul-by-0 is always safe
01016    // portable way to check for no overflows in a*b
01017    return a <= INT_MAX/b;
01018 }
01019
01020 #if !defined(STBI_NO_JPEG) || !defined(STBI_NO_PNG) || !defined(STBI_NO_TGA) || !defined(STBI_NO_HDR)
01021 // returns 1 if "a*b + add" has no negative terms/factors and doesn't overflow
01022 static int stbi__mad2sizes_valid(int a, int b, int add)
01023 {
01024    return stbi__mul2sizes_valid(a, b) && stbi__addsizes_valid(a*b, add);
01025 }
01026 #endif
01027
01028 // returns 1 if "a*b*c + add" has no negative terms/factors and doesn't overflow
01029 static int stbi__mad3sizes_valid(int a, int b, int c, int add)
01030 {
01031    return stbi__mul2sizes_valid(a, b) && stbi__mul2sizes_valid(a*b, c) &&
01032       stbi__addsizes_valid(a*b*c, add);
01033 }
01034
01035 // returns 1 if "a*b*c*d + add" has no negative terms/factors and doesn't overflow
01036 #if !defined(STBI_NO_LINEAR) || !defined(STBI_NO_HDR) || !defined(STBI_NO_PNM)
01037 static int stbi__mad4sizes_valid(int a, int b, int c, int d, int add)
01038 {
01039    return stbi__mul2sizes_valid(a, b) && stbi__mul2sizes_valid(a*b, c) &&
01040       stbi__mul2sizes_valid(a*b*c, d) && stbi__addsizes_valid(a*b*c*d, add);
01041 }
01042 #endif
01043
01044 #if !defined(STBI_NO_JPEG) || !defined(STBI_NO_PNG) || !defined(STBI_NO_TGA) || !defined(STBI_NO_HDR)
01045 // mallocs with size overflow checking
01046 static void *stbi__malloc_mad2(int a, int b, int add)
01047 {
01048    if (!stbi__mad2sizes_valid(a, b, add)) return NULL;
01049    return stbi__malloc(a*b + add);
01050 }
01051 #endif
01052
01053 static void *stbi__malloc_mad3(int a, int b, int c, int add)
01054 {
01055    if (!stbi__mad3sizes_valid(a, b, c, add)) return NULL;
01056    return stbi__malloc(a*b*c + add);
01057 }
01058
01059 #if !defined(STBI_NO_LINEAR) || !defined(STBI_NO_HDR) || !defined(STBI_NO_PNM)
01060 static void *stbi__malloc_mad4(int a, int b, int c, int d, int add)
01061 {
01062    if (!stbi__mad4sizes_valid(a, b, c, d, add)) return NULL;
01063    return stbi__malloc(a*b*c*d + add);
01064 }
01065 #endif
01066
01067 // returns 1 if the sum of two signed ints is valid (between -2^31 and 2^31-1 inclusive), 0 on
```

```
   overflow.
01068 static int stbi__addints_valid(int a, int b)
01069 {
01070    if ((a >= 0) != (b >= 0)) return 1; // a and b have different signs, so no overflow
01071    if (a < 0 && b < 0) return a >= INT_MIN - b; // same as a + b >= INT_MIN; INT_MIN - b cannot
   overflow since b < 0.
01072    return a <= INT_MAX - b;
01073 }
01074
01075 // returns 1 if the product of two signed shorts is valid, 0 on overflow.
01076 static int stbi__mul2shorts_valid(short a, short b)
01077 {
01078    if (b == 0 || b == -1) return 1; // multiplication by 0 is always 0; check for -1 so SHRT_MIN/b
   doesn't overflow
01079    if ((a >= 0) == (b >= 0)) return a <= SHRT_MAX/b; // product is positive, so similar to
   mul2sizes_valid
01080    if (b < 0) return a <= SHRT_MIN / b; // same as a * b >= SHRT_MIN
01081    return a >= SHRT_MIN / b;
01082 }
01083
01084 // stbi__err - error
01085 // stbi__errpf - error returning pointer to float
01086 // stbi__errpuc - error returning pointer to unsigned char
01087
01088 #ifdef STBI_NO_FAILURE_STRINGS
01089    #define stbi__err(x,y)  0
01090 #elif defined(STBI_FAILURE_USERMSG)
01091    #define stbi__err(x,y)  stbi__err(y)
01092 #else
01093    #define stbi__err(x,y)  stbi__err(x)
01094 #endif
01095
01096 #define stbi__errpf(x,y)   ((float *)(size_t) (stbi__err(x,y)?NULL:NULL))
01097 #define stbi__errpuc(x,y)  ((unsigned char *)(size_t) (stbi__err(x,y)?NULL:NULL))
01098
01099 STBIDEF void stbi_image_free(void *retval_from_stbi_load)
01100 {
01101    STBI_FREE(retval_from_stbi_load);
01102 }
01103
01104 #ifndef STBI_NO_LINEAR
01105 static float   *stbi__ldr_to_hdr(stbi_uc *data, int x, int y, int comp);
01106 #endif
01107
01108 #ifndef STBI_NO_HDR
01109 static stbi_uc *stbi__hdr_to_ldr(float   *data, int x, int y, int comp);
01110 #endif
01111
01112 static int stbi__vertically_flip_on_load_global = 0;
01113
01114 STBIDEF void stbi_set_flip_vertically_on_load(int flag_true_if_should_flip)
01115 {
01116    stbi__vertically_flip_on_load_global = flag_true_if_should_flip;
01117 }
01118
01119 #ifndef STBI_THREAD_LOCAL
01120 #define stbi__vertically_flip_on_load  stbi__vertically_flip_on_load_global
01121 #else
01122 static STBI_THREAD_LOCAL int stbi__vertically_flip_on_load_local, stbi__vertically_flip_on_load_set;
01123
01124 STBIDEF void stbi_set_flip_vertically_on_load_thread(int flag_true_if_should_flip)
01125 {
01126    stbi__vertically_flip_on_load_local = flag_true_if_should_flip;
01127    stbi__vertically_flip_on_load_set = 1;
01128 }
01129
01130 #define stbi__vertically_flip_on_load  (stbi__vertically_flip_on_load_set       \
01131                                          ? stbi__vertically_flip_on_load_local  \
01132                                          : stbi__vertically_flip_on_load_global)
01133 #endif // STBI_THREAD_LOCAL
01134
01135 static void *stbi__load_main(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri, int bpc)
01136 {
01137    memset(ri, 0, sizeof(*ri)); // make sure it's initialized if we add new fields
01138    ri->bits_per_channel = 8; // default is 8 so most paths don't have to be changed
01139    ri->channel_order = STBI_ORDER_RGB; // all current input & output are this, but this is here so we
   can add BGR order
01140    ri->num_channels = 0;
01141
01142    // test the formats with a very explicit header first (at least a FOURCC
01143    // or distinctive magic number first)
01144    #ifndef STBI_NO_PNG
01145    if (stbi__png_test(s))  return stbi__png_load(s,x,y,comp,req_comp, ri);
01146    #endif
01147    #ifndef STBI_NO_BMP
01148    if (stbi__bmp_test(s))  return stbi__bmp_load(s,x,y,comp,req_comp, ri);
```

```
01149    #endif
01150    #ifndef STBI_NO_GIF
01151    if (stbi__gif_test(s))  return stbi__gif_load(s,x,y,comp,req_comp, ri);
01152    #endif
01153    #ifndef STBI_NO_PSD
01154    if (stbi__psd_test(s))  return stbi__psd_load(s,x,y,comp,req_comp, ri, bpc);
01155    #else
01156    STBI_NOTUSED(bpc);
01157    #endif
01158    #ifndef STBI_NO_PIC
01159    if (stbi__pic_test(s))  return stbi__pic_load(s,x,y,comp,req_comp, ri);
01160    #endif
01161
01162    // then the formats that can end up attempting to load with just 1 or 2
01163    // bytes matching expectations; these are prone to false positives, so
01164    // try them later
01165    #ifndef STBI_NO_JPEG
01166    if (stbi__jpeg_test(s)) return stbi__jpeg_load(s,x,y,comp,req_comp, ri);
01167    #endif
01168    #ifndef STBI_NO_PNM
01169    if (stbi__pnm_test(s))  return stbi__pnm_load(s,x,y,comp,req_comp, ri);
01170    #endif
01171
01172    #ifndef STBI_NO_HDR
01173    if (stbi__hdr_test(s)) {
01174       float *hdr = stbi__hdr_load(s, x,y,comp,req_comp, ri);
01175       return stbi__hdr_to_ldr(hdr, *x, *y, req_comp ? req_comp : *comp);
01176    }
01177    #endif
01178
01179    #ifndef STBI_NO_TGA
01180    // test tga last because it's a crappy test!
01181    if (stbi__tga_test(s))
01182       return stbi__tga_load(s,x,y,comp,req_comp, ri);
01183    #endif
01184
01185    return stbi__errpuc("unknown image type", "Image not of any known type, or corrupt");
01186 }
01187
01188 static stbi_uc *stbi__convert_16_to_8(stbi__uint16 *orig, int w, int h, int channels)
01189 {
01190    int i;
01191    int img_len = w * h * channels;
01192    stbi_uc *reduced;
01193
01194    reduced = (stbi_uc *) stbi__malloc(img_len);
01195    if (reduced == NULL) return stbi__errpuc("outofmem", "Out of memory");
01196
01197    for (i = 0; i < img_len; ++i)
01198       reduced[i] = (stbi_uc)((orig[i] » 8) & 0xFF); // top half of each byte is sufficient approx of
      16->8 bit scaling
01199
01200    STBI_FREE(orig);
01201    return reduced;
01202 }
01203
01204 static stbi__uint16 *stbi__convert_8_to_16(stbi_uc *orig, int w, int h, int channels)
01205 {
01206    int i;
01207    int img_len = w * h * channels;
01208    stbi__uint16 *enlarged;
01209
01210    enlarged = (stbi__uint16 *) stbi__malloc(img_len*2);
01211    if (enlarged == NULL) return (stbi__uint16 *) stbi__errpuc("outofmem", "Out of memory");
01212
01213    for (i = 0; i < img_len; ++i)
01214       enlarged[i] = (stbi__uint16)((orig[i] « 8) + orig[i]); // replicate to high and low byte, maps
      0->0, 255->0xffff
01215
01216    STBI_FREE(orig);
01217    return enlarged;
01218 }
01219
01220 static void stbi__vertical_flip(void *image, int w, int h, int bytes_per_pixel)
01221 {
01222    int row;
01223    size_t bytes_per_row = (size_t)w * bytes_per_pixel;
01224    stbi_uc temp[2048];
01225    stbi_uc *bytes = (stbi_uc *)image;
01226
01227    for (row = 0; row < (h»1); row++) {
01228       stbi_uc *row0 = bytes + row*bytes_per_row;
01229       stbi_uc *row1 = bytes + (h - row - 1)*bytes_per_row;
01230       // swap row0 with row1
01231       size_t bytes_left = bytes_per_row;
01232       while (bytes_left) {
01233          size_t bytes_copy = (bytes_left < sizeof(temp)) ? bytes_left : sizeof(temp);
```

```
01234          memcpy(temp, row0, bytes_copy);
01235          memcpy(row0, row1, bytes_copy);
01236          memcpy(row1, temp, bytes_copy);
01237          row0 += bytes_copy;
01238          row1 += bytes_copy;
01239          bytes_left -= bytes_copy;
01240       }
01241    }
01242 }
01243
01244 #ifndef STBI_NO_GIF
01245 static void stbi__vertical_flip_slices(void *image, int w, int h, int z, int bytes_per_pixel)
01246 {
01247    int slice;
01248    int slice_size = w * h * bytes_per_pixel;
01249
01250    stbi_uc *bytes = (stbi_uc *)image;
01251    for (slice = 0; slice < z; ++slice) {
01252       stbi__vertical_flip(bytes, w, h, bytes_per_pixel);
01253       bytes += slice_size;
01254    }
01255 }
01256 #endif
01257
01258 static unsigned char *stbi__load_and_postprocess_8bit(stbi__context *s, int *x, int *y, int *comp, int
      req_comp)
01259 {
01260    stbi__result_info ri;
01261    void *result = stbi__load_main(s, x, y, comp, req_comp, &ri, 8);
01262
01263    if (result == NULL)
01264       return NULL;
01265
01266    // it is the responsibility of the loaders to make sure we get either 8 or 16 bit.
01267    STBI_ASSERT(ri.bits_per_channel == 8 || ri.bits_per_channel == 16);
01268
01269    if (ri.bits_per_channel != 8) {
01270       result = stbi__convert_16_to_8((stbi__uint16 *) result, *x, *y, req_comp == 0 ? *comp :
      req_comp);
01271       ri.bits_per_channel = 8;
01272    }
01273
01274    // @TODO: move stbi__convert_format to here
01275
01276    if (stbi__vertically_flip_on_load) {
01277       int channels = req_comp ? req_comp : *comp;
01278       stbi__vertical_flip(result, *x, *y, channels * sizeof(stbi_uc));
01279    }
01280
01281    return (unsigned char *) result;
01282 }
01283
01284 static stbi__uint16 *stbi__load_and_postprocess_16bit(stbi__context *s, int *x, int *y, int *comp, int
      req_comp)
01285 {
01286    stbi__result_info ri;
01287    void *result = stbi__load_main(s, x, y, comp, req_comp, &ri, 16);
01288
01289    if (result == NULL)
01290       return NULL;
01291
01292    // it is the responsibility of the loaders to make sure we get either 8 or 16 bit.
01293    STBI_ASSERT(ri.bits_per_channel == 8 || ri.bits_per_channel == 16);
01294
01295    if (ri.bits_per_channel != 16) {
01296       result = stbi__convert_8_to_16((stbi_uc *) result, *x, *y, req_comp == 0 ? *comp : req_comp);
01297       ri.bits_per_channel = 16;
01298    }
01299
01300    // @TODO: move stbi__convert_format16 to here
01301    // @TODO: special case RGB-to-Y (and RGBA-to-YA) for 8-bit-to-16-bit case to keep more precision
01302
01303    if (stbi__vertically_flip_on_load) {
01304       int channels = req_comp ? req_comp : *comp;
01305       stbi__vertical_flip(result, *x, *y, channels * sizeof(stbi__uint16));
01306    }
01307
01308    return (stbi__uint16 *) result;
01309 }
01310
01311 #if !defined(STBI_NO_HDR) && !defined(STBI_NO_LINEAR)
01312 static void stbi__float_postprocess(float *result, int *x, int *y, int *comp, int req_comp)
01313 {
01314    if (stbi__vertically_flip_on_load && result != NULL) {
01315       int channels = req_comp ? req_comp : *comp;
01316       stbi__vertical_flip(result, *x, *y, channels * sizeof(float));
01317    }
```

```
01318 }
01319 #endif
01320
01321 #ifndef STBI_NO_STDIO
01322
01323 #if defined(_WIN32) && defined(STBI_WINDOWS_UTF8)
01324 STBI_EXTERN __declspec(dllimport) int __stdcall MultiByteToWideChar(unsigned int cp, unsigned long
      flags, const char *str, int cbmb, wchar_t *widestr, int cchwide);
01325 STBI_EXTERN __declspec(dllimport) int __stdcall WideCharToMultiByte(unsigned int cp, unsigned long
      flags, const wchar_t *widestr, int cchwide, char *str, int cbmb, const char *defchar, int
      *used_default);
01326 #endif
01327
01328 #if defined(_WIN32) && defined(STBI_WINDOWS_UTF8)
01329 STBIDEF int stbi_convert_wchar_to_utf8(char *buffer, size_t bufferlen, const wchar_t* input)
01330 {
01331     return WideCharToMultiByte(65001 /* UTF8 */, 0, input, -1, buffer, (int) bufferlen, NULL, NULL);
01332 }
01333 #endif
01334
01335 static FILE *stbi__fopen(char const *filename, char const *mode)
01336 {
01337    FILE *f;
01338 #if defined(_WIN32) && defined(STBI_WINDOWS_UTF8)
01339    wchar_t wMode[64];
01340    wchar_t wFilename[1024];
01341     if (0 == MultiByteToWideChar(65001 /* UTF8 */, 0, filename, -1, wFilename,
      sizeof(wFilename)/sizeof(*wFilename)))
01342       return 0;
01343
01344     if (0 == MultiByteToWideChar(65001 /* UTF8 */, 0, mode, -1, wMode, sizeof(wMode)/sizeof(*wMode)))
01345       return 0;
01346
01347 #if defined(_MSC_VER) && _MSC_VER >= 1400
01348     if (0 != _wfopen_s(&f, wFilename, wMode))
01349        f = 0;
01350 #else
01351    f = _wfopen(wFilename, wMode);
01352 #endif
01353
01354 #elif defined(_MSC_VER) && _MSC_VER >= 1400
01355    if (0 != fopen_s(&f, filename, mode))
01356       f=0;
01357 #else
01358    f = fopen(filename, mode);
01359 #endif
01360    return f;
01361 }
01362
01363
01364 STBIDEF stbi_uc *stbi_load(char const *filename, int *x, int *y, int *comp, int req_comp)
01365 {
01366    FILE *f = stbi__fopen(filename, "rb");
01367    unsigned char *result;
01368    if (!f) return stbi__errpuc("can't fopen", "Unable to open file");
01369    result = stbi_load_from_file(f,x,y,comp,req_comp);
01370    fclose(f);
01371    return result;
01372 }
01373
01374 STBIDEF stbi_uc *stbi_load_from_file(FILE *f, int *x, int *y, int *comp, int req_comp)
01375 {
01376    unsigned char *result;
01377    stbi__context s;
01378    stbi__start_file(&s,f);
01379    result = stbi__load_and_postprocess_8bit(&s,x,y,comp,req_comp);
01380    if (result) {
01381       // need to 'unget' all the characters in the IO buffer
01382       fseek(f, - (int) (s.img_buffer_end - s.img_buffer), SEEK_CUR);
01383    }
01384    return result;
01385 }
01386
01387 STBIDEF stbi__uint16 *stbi_load_from_file_16(FILE *f, int *x, int *y, int *comp, int req_comp)
01388 {
01389    stbi__uint16 *result;
01390    stbi__context s;
01391    stbi__start_file(&s,f);
01392    result = stbi__load_and_postprocess_16bit(&s,x,y,comp,req_comp);
01393    if (result) {
01394       // need to 'unget' all the characters in the IO buffer
01395       fseek(f, - (int) (s.img_buffer_end - s.img_buffer), SEEK_CUR);
01396    }
01397    return result;
01398 }
01399
01400 STBIDEF stbi_us *stbi_load_16(char const *filename, int *x, int *y, int *comp, int req_comp)
```

```
01401 {
01402    FILE *f = stbi__fopen(filename, "rb");
01403    stbi__uint16 *result;
01404    if (!f) return (stbi_us *) stbi__errpuc("can't fopen", "Unable to open file");
01405    result = stbi_load_from_file_16(f,x,y,comp,req_comp);
01406    fclose(f);
01407    return result;
01408 }
01409
01410
01411 #endif
01412
01413 STBIDEF stbi_us *stbi_load_16_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int
      *channels_in_file, int desired_channels)
01414 {
01415    stbi__context s;
01416    stbi__start_mem(&s,buffer,len);
01417    return stbi__load_and_postprocess_16bit(&s,x,y,channels_in_file,desired_channels);
01418 }
01419
01420 STBIDEF stbi_us *stbi_load_16_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int
      *y, int *channels_in_file, int desired_channels)
01421 {
01422    stbi__context s;
01423    stbi__start_callbacks(&s, (stbi_io_callbacks *)clbk, user);
01424    return stbi__load_and_postprocess_16bit(&s,x,y,channels_in_file,desired_channels);
01425 }
01426
01427 STBIDEF stbi_uc *stbi_load_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp, int
      req_comp)
01428 {
01429    stbi__context s;
01430    stbi__start_mem(&s,buffer,len);
01431    return stbi__load_and_postprocess_8bit(&s,x,y,comp,req_comp);
01432 }
01433
01434 STBIDEF stbi_uc *stbi_load_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y,
      int *comp, int req_comp)
01435 {
01436    stbi__context s;
01437    stbi__start_callbacks(&s, (stbi_io_callbacks *) clbk, user);
01438    return stbi__load_and_postprocess_8bit(&s,x,y,comp,req_comp);
01439 }
01440
01441 #ifndef STBI_NO_GIF
01442 STBIDEF stbi_uc *stbi_load_gif_from_memory(stbi_uc const *buffer, int len, int **delays, int *x, int
      *y, int *z, int *comp, int req_comp)
01443 {
01444    unsigned char *result;
01445    stbi__context s;
01446    stbi__start_mem(&s,buffer,len);
01447
01448    result = (unsigned char*) stbi__load_gif_main(&s, delays, x, y, z, comp, req_comp);
01449    if (stbi__vertically_flip_on_load) {
01450       stbi__vertical_flip_slices( result, *x, *y, *z, *comp );
01451    }
01452
01453    return result;
01454 }
01455 #endif
01456
01457 #ifndef STBI_NO_LINEAR
01458 static float *stbi__loadf_main(stbi__context *s, int *x, int *y, int *comp, int req_comp)
01459 {
01460    unsigned char *data;
01461    #ifndef STBI_NO_HDR
01462    if (stbi__hdr_test(s)) {
01463       stbi__result_info ri;
01464       float *hdr_data = stbi__hdr_load(s,x,y,comp,req_comp, &ri);
01465       if (hdr_data)
01466          stbi__float_postprocess(hdr_data,x,y,comp,req_comp);
01467       return hdr_data;
01468    }
01469    #endif
01470    data = stbi__load_and_postprocess_8bit(s, x, y, comp, req_comp);
01471    if (data)
01472       return stbi__ldr_to_hdr(data, *x, *y, req_comp ? req_comp : *comp);
01473    return stbi__errpf("unknown image type", "Image not of any known type, or corrupt");
01474 }
01475
01476 STBIDEF float *stbi_loadf_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp, int
      req_comp)
01477 {
01478    stbi__context s;
01479    stbi__start_mem(&s,buffer,len);
01480    return stbi__loadf_main(&s,x,y,comp,req_comp);
01481 }
```

```
01482
01483 STBIDEF float *stbi_loadf_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y,
       int *comp, int req_comp)
01484 {
01485    stbi__context s;
01486    stbi__start_callbacks(&s, (stbi_io_callbacks *) clbk, user);
01487    return stbi__loadf_main(&s,x,y,comp,req_comp);
01488 }
01489
01490 #ifndef STBI_NO_STDIO
01491 STBIDEF float *stbi_loadf(char const *filename, int *x, int *y, int *comp, int req_comp)
01492 {
01493    float *result;
01494    FILE *f = stbi__fopen(filename, "rb");
01495    if (!f) return stbi__errpf("can't fopen", "Unable to open file");
01496    result = stbi_loadf_from_file(f,x,y,comp,req_comp);
01497    fclose(f);
01498    return result;
01499 }
01500
01501 STBIDEF float *stbi_loadf_from_file(FILE *f, int *x, int *y, int *comp, int req_comp)
01502 {
01503    stbi__context s;
01504    stbi__start_file(&s,f);
01505    return stbi__loadf_main(&s,x,y,comp,req_comp);
01506 }
01507 #endif // !STBI_NO_STDIO
01508
01509 #endif // !STBI_NO_LINEAR
01510
01511 // these is-hdr-or-not is defined independent of whether STBI_NO_LINEAR is
01512 // defined, for API simplicity; if STBI_NO_LINEAR is defined, it always
01513 // reports false!
01514
01515 STBIDEF int stbi_is_hdr_from_memory(stbi_uc const *buffer, int len)
01516 {
01517    #ifndef STBI_NO_HDR
01518    stbi__context s;
01519    stbi__start_mem(&s,buffer,len);
01520    return stbi__hdr_test(&s);
01521    #else
01522    STBI_NOTUSED(buffer);
01523    STBI_NOTUSED(len);
01524    return 0;
01525    #endif
01526 }
01527
01528 #ifndef STBI_NO_STDIO
01529 STBIDEF int      stbi_is_hdr          (char const *filename)
01530 {
01531    FILE *f = stbi__fopen(filename, "rb");
01532    int result=0;
01533    if (f) {
01534       result = stbi_is_hdr_from_file(f);
01535       fclose(f);
01536    }
01537    return result;
01538 }
01539
01540 STBIDEF int stbi_is_hdr_from_file(FILE *f)
01541 {
01542    #ifndef STBI_NO_HDR
01543    long pos = ftell(f);
01544    int res;
01545    stbi__context s;
01546    stbi__start_file(&s,f);
01547    res = stbi__hdr_test(&s);
01548    fseek(f, pos, SEEK_SET);
01549    return res;
01550    #else
01551    STBI_NOTUSED(f);
01552    return 0;
01553    #endif
01554 }
01555 #endif // !STBI_NO_STDIO
01556
01557 STBIDEF int      stbi_is_hdr_from_callbacks(stbi_io_callbacks const *clbk, void *user)
01558 {
01559    #ifndef STBI_NO_HDR
01560    stbi__context s;
01561    stbi__start_callbacks(&s, (stbi_io_callbacks *) clbk, user);
01562    return stbi__hdr_test(&s);
01563    #else
01564    STBI_NOTUSED(clbk);
01565    STBI_NOTUSED(user);
01566    return 0;
01567    #endif
```

```
01568 }
01569
01570 #ifndef STBI_NO_LINEAR
01571 static float stbi__l2h_gamma=2.2f, stbi__l2h_scale=1.0f;
01572
01573 STBIDEF void   stbi_ldr_to_hdr_gamma(float gamma) { stbi__l2h_gamma = gamma; }
01574 STBIDEF void   stbi_ldr_to_hdr_scale(float scale) { stbi__l2h_scale = scale; }
01575 #endif
01576
01577 static float stbi__h2l_gamma_i=1.0f/2.2f, stbi__h2l_scale_i=1.0f;
01578
01579 STBIDEF void   stbi_hdr_to_ldr_gamma(float gamma) { stbi__h2l_gamma_i = 1/gamma; }
01580 STBIDEF void   stbi_hdr_to_ldr_scale(float scale) { stbi__h2l_scale_i = 1/scale; }
01581
01582
01584 //
01585 // Common code used by all image loaders
01586 //
01587
01588 enum
01589 {
01590    STBI__SCAN_load=0,
01591    STBI__SCAN_type,
01592    STBI__SCAN_header
01593 };
01594
01595 static void stbi__refill_buffer(stbi__context *s)
01596 {
01597    int n = (s->io.read)(s->io_user_data,(char*)s->buffer_start,s->buflen);
01598    s->callback_already_read += (int) (s->img_buffer - s->img_buffer_original);
01599    if (n == 0) {
01600       // at end of file, treat same as if from memory, but need to handle case
01601       // where s->img_buffer isn't pointing to safe memory, e.g. 0-byte file
01602       s->read_from_callbacks = 0;
01603       s->img_buffer = s->buffer_start;
01604       s->img_buffer_end = s->buffer_start+1;
01605       *s->img_buffer = 0;
01606    } else {
01607       s->img_buffer = s->buffer_start;
01608       s->img_buffer_end = s->buffer_start + n;
01609    }
01610 }
01611
01612 stbi_inline static stbi_uc stbi__get8(stbi__context *s)
01613 {
01614    if (s->img_buffer < s->img_buffer_end)
01615       return *s->img_buffer++;
01616    if (s->read_from_callbacks) {
01617       stbi__refill_buffer(s);
01618       return *s->img_buffer++;
01619    }
01620    return 0;
01621 }
01622
01623 #if defined(STBI_NO_JPEG) && defined(STBI_NO_HDR) && defined(STBI_NO_PIC) && defined(STBI_NO_PNM)
01624 // nothing
01625 #else
01626 stbi_inline static int stbi__at_eof(stbi__context *s)
01627 {
01628    if (s->io.read) {
01629       if (!(s->io.eof)(s->io_user_data)) return 0;
01630       // if feof() is true, check if buffer = end
01631       // special case: we've only got the special 0 character at the end
01632       if (s->read_from_callbacks == 0) return 1;
01633    }
01634
01635    return s->img_buffer >= s->img_buffer_end;
01636 }
01637 #endif
01638
01639 #if defined(STBI_NO_JPEG) && defined(STBI_NO_PNG) && defined(STBI_NO_BMP) && defined(STBI_NO_PSD) &&
      defined(STBI_NO_TGA) && defined(STBI_NO_GIF) && defined(STBI_NO_PIC)
01640 // nothing
01641 #else
01642 static void stbi__skip(stbi__context *s, int n)
01643 {
01644    if (n == 0) return;  // already there!
01645    if (n < 0) {
01646       s->img_buffer = s->img_buffer_end;
01647       return;
01648    }
01649    if (s->io.read) {
01650       int blen = (int) (s->img_buffer_end - s->img_buffer);
01651       if (blen < n) {
01652          s->img_buffer = s->img_buffer_end;
01653          (s->io.skip)(s->io_user_data, n - blen);
01654          return;
```

```
01655       }
01656    }
01657    s->img_buffer += n;
01658 }
01659 #endif
01660
01661 #if defined(STBI_NO_PNG) && defined(STBI_NO_TGA) && defined(STBI_NO_HDR) && defined(STBI_NO_PNM)
01662 // nothing
01663 #else
01664 static int stbi__getn(stbi__context *s, stbi_uc *buffer, int n)
01665 {
01666    if (s->io.read) {
01667       int blen = (int) (s->img_buffer_end - s->img_buffer);
01668       if (blen < n) {
01669          int res, count;
01670
01671          memcpy(buffer, s->img_buffer, blen);
01672
01673          count = (s->io.read)(s->io_user_data, (char*) buffer + blen, n - blen);
01674          res = (count == (n-blen));
01675          s->img_buffer = s->img_buffer_end;
01676          return res;
01677       }
01678    }
01679
01680    if (s->img_buffer+n <= s->img_buffer_end) {
01681       memcpy(buffer, s->img_buffer, n);
01682       s->img_buffer += n;
01683       return 1;
01684    } else
01685       return 0;
01686 }
01687 #endif
01688
01689 #if defined(STBI_NO_JPEG) && defined(STBI_NO_PNG) && defined(STBI_NO_PSD) && defined(STBI_NO_PIC)
01690 // nothing
01691 #else
01692 static int stbi__get16be(stbi__context *s)
01693 {
01694    int z = stbi__get8(s);
01695    return (z << 8) + stbi__get8(s);
01696 }
01697 #endif
01698
01699 #if defined(STBI_NO_PNG) && defined(STBI_NO_PSD) && defined(STBI_NO_PIC)
01700 // nothing
01701 #else
01702 static stbi__uint32 stbi__get32be(stbi__context *s)
01703 {
01704    stbi__uint32 z = stbi__get16be(s);
01705    return (z << 16) + stbi__get16be(s);
01706 }
01707 #endif
01708
01709 #if defined(STBI_NO_BMP) && defined(STBI_NO_TGA) && defined(STBI_NO_GIF)
01710 // nothing
01711 #else
01712 static int stbi__get16le(stbi__context *s)
01713 {
01714    int z = stbi__get8(s);
01715    return z + (stbi__get8(s) << 8);
01716 }
01717 #endif
01718
01719 #ifndef STBI_NO_BMP
01720 static stbi__uint32 stbi__get32le(stbi__context *s)
01721 {
01722    stbi__uint32 z = stbi__get16le(s);
01723    z += (stbi__uint32)stbi__get16le(s) << 16;
01724    return z;
01725 }
01726 #endif
01727
01728 #define STBI__BYTECAST(x)  ((stbi_uc) ((x) & 255))  // truncate int to byte without warnings
01729
01730 #if defined(STBI_NO_JPEG) && defined(STBI_NO_PNG) && defined(STBI_NO_BMP) && defined(STBI_NO_PSD) &&
      defined(STBI_NO_TGA) && defined(STBI_NO_GIF) && defined(STBI_NO_PIC) && defined(STBI_NO_PNM)
01731 // nothing
01732 #else
01734 //
01735 //  generic converter from built-in img_n to req_comp
01736 //    individual types do this automatically as much as possible (e.g. jpeg
01737 //    does all cases internally since it needs to colorspace convert anyway,
01738 //    and it never has alpha, so very few cases ). png can automatically
01739 //    interleave an alpha=255 channel, but falls back to this for other cases
01740 //
01741 //  assume data buffer is malloced, so malloc a new one and free that one
```

```
01742 //  only failure mode is malloc failing
01743
01744 static stbi_uc stbi__compute_y(int r, int g, int b)
01745 {
01746    return (stbi_uc) (((r*77) + (g*150) +  (29*b)) » 8);
01747 }
01748 #endif
01749
01750 #if defined(STBI_NO_PNG) && defined(STBI_NO_BMP) && defined(STBI_NO_PSD) && defined(STBI_NO_TGA) &&
      defined(STBI_NO_GIF) && defined(STBI_NO_PIC) && defined(STBI_NO_PNM)
01751 // nothing
01752 #else
01753 static unsigned char *stbi__convert_format(unsigned char *data, int img_n, int req_comp, unsigned int
      x, unsigned int y)
01754 {
01755    int i,j;
01756    unsigned char *good;
01757
01758    if (req_comp == img_n) return data;
01759    STBI_ASSERT(req_comp >= 1 && req_comp <= 4);
01760
01761    good = (unsigned char *) stbi__malloc_mad3(req_comp, x, y, 0);
01762    if (good == NULL) {
01763       STBI_FREE(data);
01764       return stbi__errpuc("outofmem", "Out of memory");
01765    }
01766
01767    for (j=0; j < (int) y; ++j) {
01768       unsigned char *src  = data + j * x * img_n   ;
01769       unsigned char *dest = good + j * x * req_comp;
01770
01771       #define STBI__COMBO(a,b)  ((a)*8+(b))
01772       #define STBI__CASE(a,b)   case STBI__COMBO(a,b): for(i=x-1; i >= 0; --i, src += a, dest += b)
01773       // convert source image with img_n components to one with req_comp components;
01774       // avoid switch per pixel, so use switch per scanline and massive macros
01775       switch (STBI__COMBO(img_n, req_comp)) {
01776          STBI__CASE(1,2) { dest[0]=src[0]; dest[1]=255;                                     } break;
01777          STBI__CASE(1,3) { dest[0]=dest[1]=dest[2]=src[0];                                  } break;
01778          STBI__CASE(1,4) { dest[0]=dest[1]=dest[2]=src[0]; dest[3]=255;                     } break;
01779          STBI__CASE(2,1) { dest[0]=src[0];                                                  } break;
01780          STBI__CASE(2,3) { dest[0]=dest[1]=dest[2]=src[0];                                  } break;
01781          STBI__CASE(2,4) { dest[0]=dest[1]=dest[2]=src[0]; dest[3]=src[1];                  } break;
01782          STBI__CASE(3,4) { dest[0]=src[0];dest[1]=src[1];dest[2]=src[2];dest[3]=255;        } break;
01783          STBI__CASE(3,1) { dest[0]=stbi__compute_y(src[0],src[1],src[2]);                   } break;
01784          STBI__CASE(3,2) { dest[0]=stbi__compute_y(src[0],src[1],src[2]); dest[1] = 255;    } break;
01785          STBI__CASE(4,1) { dest[0]=stbi__compute_y(src[0],src[1],src[2]);                   } break;
01786          STBI__CASE(4,2) { dest[0]=stbi__compute_y(src[0],src[1],src[2]); dest[1] = src[3]; } break;
01787          STBI__CASE(4,3) { dest[0]=src[0];dest[1]=src[1];dest[2]=src[2];                    } break;
01788          default: STBI_ASSERT(0); STBI_FREE(data); STBI_FREE(good); return stbi__errpuc("unsupported",
      "Unsupported format conversion");
01789       }
01790       #undef STBI__CASE
01791    }
01792
01793    STBI_FREE(data);
01794    return good;
01795 }
01796 #endif
01797
01798 #if defined(STBI_NO_PNG) && defined(STBI_NO_PSD)
01799 // nothing
01800 #else
01801 static stbi__uint16 stbi__compute_y_16(int r, int g, int b)
01802 {
01803    return (stbi__uint16) (((r*77) + (g*150) +  (29*b)) » 8);
01804 }
01805 #endif
01806
01807 #if defined(STBI_NO_PNG) && defined(STBI_NO_PSD)
01808 // nothing
01809 #else
01810 static stbi__uint16 *stbi__convert_format16(stbi__uint16 *data, int img_n, int req_comp, unsigned int
      x, unsigned int y)
01811 {
01812    int i,j;
01813    stbi__uint16 *good;
01814
01815    if (req_comp == img_n) return data;
01816    STBI_ASSERT(req_comp >= 1 && req_comp <= 4);
01817
01818    good = (stbi__uint16 *) stbi__malloc(req_comp * x * y * 2);
01819    if (good == NULL) {
01820       STBI_FREE(data);
01821       return (stbi__uint16 *) stbi__errpuc("outofmem", "Out of memory");
01822    }
01823
01824    for (j=0; j < (int) y; ++j) {
```

```
01825       stbi__uint16 *src  = data + j * x * img_n   ;
01826       stbi__uint16 *dest = good + j * x * req_comp;
01827
01828       #define STBI__COMBO(a,b)  ((a)*8+(b))
01829       #define STBI__CASE(a,b)   case STBI__COMBO(a,b): for(i=x-1; i >= 0; --i, src += a, dest += b)
01830       // convert source image with img_n components to one with req_comp components;
01831       // avoid switch per pixel, so use switch per scanline and massive macros
01832       switch (STBI__COMBO(img_n, req_comp)) {
01833          STBI__CASE(1,2) { dest[0]=src[0]; dest[1]=0xffff;                                     }
      break;
01834          STBI__CASE(1,3) { dest[0]=dest[1]=dest[2]=src[0];                                     }
      break;
01835          STBI__CASE(1,4) { dest[0]=dest[1]=dest[2]=src[0]; dest[3]=0xffff;                     }
      break;
01836          STBI__CASE(2,1) { dest[0]=src[0];                                                     }
      break;
01837          STBI__CASE(2,3) { dest[0]=dest[1]=dest[2]=src[0];                                     }
      break;
01838          STBI__CASE(2,4) { dest[0]=dest[1]=dest[2]=src[0]; dest[3]=src[1];                     }
      break;
01839          STBI__CASE(3,4) { dest[0]=src[0];dest[1]=src[1];dest[2]=src[2];dest[3]=0xffff;        }
      break;
01840          STBI__CASE(3,1) { dest[0]=stbi__compute_y_16(src[0],src[1],src[2]);                   }
      break;
01841          STBI__CASE(3,2) { dest[0]=stbi__compute_y_16(src[0],src[1],src[2]); dest[1] = 0xffff; }
      break;
01842          STBI__CASE(4,1) { dest[0]=stbi__compute_y_16(src[0],src[1],src[2]);                   }
      break;
01843          STBI__CASE(4,2) { dest[0]=stbi__compute_y_16(src[0],src[1],src[2]); dest[1] = src[3]; }
      break;
01844          STBI__CASE(4,3) { dest[0]=src[0];dest[1]=src[1];dest[2]=src[2];                       }
      break;
01845          default: STBI_ASSERT(0); STBI_FREE(data); STBI_FREE(good); return (stbi__uint16*)
      stbi__errpuc("unsupported", "Unsupported format conversion");
01846       }
01847       #undef STBI__CASE
01848    }
01849
01850    STBI_FREE(data);
01851    return good;
01852 }
01853 #endif
01854
01855 #ifndef STBI_NO_LINEAR
01856 static float   *stbi__ldr_to_hdr(stbi_uc *data, int x, int y, int comp)
01857 {
01858    int i,k,n;
01859    float *output;
01860    if (!data) return NULL;
01861    output = (float *) stbi__malloc_mad4(x, y, comp, sizeof(float), 0);
01862    if (output == NULL) { STBI_FREE(data); return stbi__errpf("outofmem", "Out of memory"); }
01863    // compute number of non-alpha components
01864    if (comp & 1) n = comp; else n = comp-1;
01865    for (i=0; i < x*y; ++i) {
01866       for (k=0; k < n; ++k) {
01867          output[i*comp + k] = (float) (pow(data[i*comp+k]/255.0f, stbi__l2h_gamma) * stbi__l2h_scale);
01868       }
01869    }
01870    if (n < comp) {
01871       for (i=0; i < x*y; ++i) {
01872          output[i*comp + n] = data[i*comp + n]/255.0f;
01873       }
01874    }
01875    STBI_FREE(data);
01876    return output;
01877 }
01878 #endif
01879
01880 #ifndef STBI_NO_HDR
01881 #define stbi__float2int(x)   ((int) (x))
01882 static stbi_uc *stbi__hdr_to_ldr(float   *data, int x, int y, int comp)
01883 {
01884    int i,k,n;
01885    stbi_uc *output;
01886    if (!data) return NULL;
01887    output = (stbi_uc *) stbi__malloc_mad3(x, y, comp, 0);
01888    if (output == NULL) { STBI_FREE(data); return stbi__errpuc("outofmem", "Out of memory"); }
01889    // compute number of non-alpha components
01890    if (comp & 1) n = comp; else n = comp-1;
01891    for (i=0; i < x*y; ++i) {
01892       for (k=0; k < n; ++k) {
01893          float z = (float) pow(data[i*comp+k]*stbi__h2l_scale_i, stbi__h2l_gamma_i) * 255 + 0.5f;
01894          if (z < 0) z = 0;
01895          if (z > 255) z = 255;
01896          output[i*comp + k] = (stbi_uc) stbi__float2int(z);
01897       }
01898       if (k < comp) {
```

```
01899          float z = data[i*comp+k] * 255 + 0.5f;
01900          if (z < 0) z = 0;
01901          if (z > 255) z = 255;
01902          output[i*comp + k] = (stbi_uc) stbi__float2int(z);
01903       }
01904    }
01905    STBI_FREE(data);
01906    return output;
01907 }
01908 #endif
01909
01911 //
01912 //  "baseline" JPEG/JFIF decoder
01913 //
01914 //    simple implementation
01915 //      - doesn't support delayed output of y-dimension
01916 //      - simple interface (only one output format: 8-bit interleaved RGB)
01917 //      - doesn't try to recover corrupt jpegs
01918 //      - doesn't allow partial loading, loading multiple at once
01919 //      - still fast on x86 (copying globals into locals doesn't help x86)
01920 //      - allocates lots of intermediate memory (full size of all components)
01921 //        - non-interleaved case requires this anyway
01922 //        - allows good upsampling (see next)
01923 //    high-quality
01924 //      - upsampled channels are bilinearly interpolated, even across blocks
01925 //      - quality integer IDCT derived from IJG's 'slow'
01926 //    performance
01927 //      - fast huffman; reasonable integer IDCT
01928 //      - some SIMD kernels for common paths on targets with SSE2/NEON
01929 //      - uses a lot of intermediate memory, could cache poorly
01930
01931 #ifndef STBI_NO_JPEG
01932
01933 // huffman decoding acceleration
01934 #define FAST_BITS   9  // larger handles more cases; smaller stomps less cache
01935
01936 typedef struct
01937 {
01938    stbi_uc  fast[1 << FAST_BITS];
01939    // weirdly, repacking this into AoS is a 10% speed loss, instead of a win
01940    stbi__uint16 code[256];
01941    stbi_uc  values[256];
01942    stbi_uc  size[257];
01943    unsigned int maxcode[18];
01944    int    delta[17];   // old 'firstsymbol' - old 'firstcode'
01945 } stbi__huffman;
01946
01947 typedef struct
01948 {
01949    stbi__context *s;
01950    stbi__huffman huff_dc[4];
01951    stbi__huffman huff_ac[4];
01952    stbi__uint16 dequant[4][64];
01953    stbi__int16 fast_ac[4][1 << FAST_BITS];
01954
01955 // sizes for components, interleaved MCUs
01956    int img_h_max, img_v_max;
01957    int img_mcu_x, img_mcu_y;
01958    int img_mcu_w, img_mcu_h;
01959
01960 // definition of jpeg image component
01961    struct
01962    {
01963       int id;
01964       int h,v;
01965       int tq;
01966       int hd,ha;
01967       int dc_pred;
01968
01969       int x,y,w2,h2;
01970       stbi_uc *data;
01971       void *raw_data, *raw_coeff;
01972       stbi_uc *linebuf;
01973       short   *coeff;   // progressive only
01974       int      coeff_w, coeff_h; // number of 8x8 coefficient blocks
01975    } img_comp[4];
01976
01977    stbi__uint32   code_buffer; // jpeg entropy-coded buffer
01978    int            code_bits;   // number of valid bits
01979    unsigned char  marker;      // marker seen while filling entropy buffer
01980    int            nomore;      // flag if we saw a marker so must stop
01981
01982    int            progressive;
01983    int            spec_start;
01984    int            spec_end;
01985    int            succ_high;
01986    int            succ_low;
```

```
01987    int            eob_run;
01988    int            jfif;
01989    int            app14_color_transform; // Adobe APP14 tag
01990    int            rgb;
01991
01992    int scan_n, order[4];
01993    int restart_interval, todo;
01994
01995 // kernels
01996    void (*idct_block_kernel)(stbi_uc *out, int out_stride, short data[64]);
01997    void (*YCbCr_to_RGB_kernel)(stbi_uc *out, const stbi_uc *y, const stbi_uc *pcb, const stbi_uc *pcr,
       int count, int step);
01998    stbi_uc *(*resample_row_hv_2_kernel)(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int
       hs);
01999 } stbi__jpeg;
02000
02001 static int stbi__build_huffman(stbi__huffman *h, int *count)
02002 {
02003    int i,j,k=0;
02004    unsigned int code;
02005    // build size list for each symbol (from JPEG spec)
02006    for (i=0; i < 16; ++i) {
02007       for (j=0; j < count[i]; ++j) {
02008          h->size[k++] = (stbi_uc) (i+1);
02009          if(k >= 257) return stbi__err("bad size list","Corrupt JPEG");
02010       }
02011    }
02012    h->size[k] = 0;
02013
02014    // compute actual symbols (from jpeg spec)
02015    code = 0;
02016    k = 0;
02017    for(j=1; j <= 16; ++j) {
02018       // compute delta to add to code to compute symbol id
02019       h->delta[j] = k - code;
02020       if (h->size[k] == j) {
02021          while (h->size[k] == j)
02022             h->code[k++] = (stbi__uint16) (code++);
02023          if (code-1 >= (1u << j)) return stbi__err("bad code lengths","Corrupt JPEG");
02024       }
02025       // compute largest code + 1 for this size, preshifted as needed later
02026       h->maxcode[j] = code << (16-j);
02027       code <<= 1;
02028    }
02029    h->maxcode[j] = 0xffffffff;
02030
02031    // build non-spec acceleration table; 255 is flag for not-accelerated
02032    memset(h->fast, 255, 1 << FAST_BITS);
02033    for (i=0; i < k; ++i) {
02034       int s = h->size[i];
02035       if (s <= FAST_BITS) {
02036          int c = h->code[i] << (FAST_BITS-s);
02037          int m = 1 << (FAST_BITS-s);
02038          for (j=0; j < m; ++j) {
02039             h->fast[c+j] = (stbi_uc) i;
02040          }
02041       }
02042    }
02043    return 1;
02044 }
02045
02046 // build a table that decodes both magnitude and value of small ACs in
02047 // one go.
02048 static void stbi__build_fast_ac(stbi__int16 *fast_ac, stbi__huffman *h)
02049 {
02050    int i;
02051    for (i=0; i < (1 << FAST_BITS); ++i) {
02052       stbi_uc fast = h->fast[i];
02053       fast_ac[i] = 0;
02054       if (fast < 255) {
02055          int rs = h->values[fast];
02056          int run = (rs >> 4) & 15;
02057          int magbits = rs & 15;
02058          int len = h->size[fast];
02059
02060          if (magbits && len + magbits <= FAST_BITS) {
02061             // magnitude code followed by receive_extend code
02062             int k = ((i << len) & ((1 << FAST_BITS) - 1)) >> (FAST_BITS - magbits);
02063             int m = 1 << (magbits - 1);
02064             if (k < m) k += (~0U << magbits) + 1;
02065             // if the result is small enough, we can fit it in fast_ac table
02066             if (k >= -128 && k <= 127)
02067                fast_ac[i] = (stbi__int16) ((k * 256) + (run * 16) + (len + magbits));
02068          }
02069       }
02070    }
02071 }
```

```
02072
02073 static void stbi__grow_buffer_unsafe(stbi__jpeg *j)
02074 {
02075    do {
02076       unsigned int b = j->nomore ? 0 : stbi__get8(j->s);
02077       if (b == 0xff) {
02078          int c = stbi__get8(j->s);
02079          while (c == 0xff) c = stbi__get8(j->s); // consume fill bytes
02080          if (c != 0) {
02081             j->marker = (unsigned char) c;
02082             j->nomore = 1;
02083             return;
02084          }
02085       }
02086       j->code_buffer |= b << (24 - j->code_bits);
02087       j->code_bits += 8;
02088    } while (j->code_bits <= 24);
02089 }
02090
02091 // (1 << n) - 1
02092 static const stbi__uint32
      stbi__bmask[17]={0,1,3,7,15,31,63,127,255,511,1023,2047,4095,8191,16383,32767,65535};
02093
02094 // decode a jpeg huffman value from the bitstream
02095 stbi_inline static int stbi__jpeg_huff_decode(stbi__jpeg *j, stbi__huffman *h)
02096 {
02097    unsigned int temp;
02098    int c,k;
02099
02100    if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
02101
02102    // look at the top FAST_BITS and determine what symbol ID it is,
02103    // if the code is <= FAST_BITS
02104    c = (j->code_buffer >> (32 - FAST_BITS)) & ((1 << FAST_BITS)-1);
02105    k = h->fast[c];
02106    if (k < 255) {
02107       int s = h->size[k];
02108       if (s > j->code_bits)
02109          return -1;
02110       j->code_buffer <<= s;
02111       j->code_bits -= s;
02112       return h->values[k];
02113    }
02114
02115    // naive test is to shift the code_buffer down so k bits are
02116    // valid, then test against maxcode. To speed this up, we've
02117    // preshifted maxcode left so that it has (16-k) 0s at the
02118    // end; in other words, regardless of the number of bits, it
02119    // wants to be compared against something shifted to have 16;
02120    // that way we don't need to shift inside the loop.
02121    temp = j->code_buffer >> 16;
02122    for (k=FAST_BITS+1 ; ; ++k)
02123       if (temp < h->maxcode[k])
02124          break;
02125    if (k == 17) {
02126       // error! code not found
02127       j->code_bits -= 16;
02128       return -1;
02129    }
02130
02131    if (k > j->code_bits)
02132       return -1;
02133
02134    // convert the huffman code to the symbol id
02135    c = ((j->code_buffer >> (32 - k)) & stbi__bmask[k]) + h->delta[k];
02136    if(c < 0 || c >= 256) // symbol id out of bounds!
02137        return -1;
02138    STBI_ASSERT((((j->code_buffer) >> (32 - h->size[c])) & stbi__bmask[h->size[c]]) == h->code[c]);
02139
02140    // convert the id to a symbol
02141    j->code_bits -= k;
02142    j->code_buffer <<= k;
02143    return h->values[c];
02144 }
02145
02146 // bias[n] = (-1<<n) + 1
02147 static const int stbi__jbias[16] =
      {0,-1,-3,-7,-15,-31,-63,-127,-255,-511,-1023,-2047,-4095,-8191,-16383,-32767};
02148
02149 // combined JPEG 'receive' and JPEG 'extend', since baseline
02150 // always extends everything it receives.
02151 stbi_inline static int stbi__extend_receive(stbi__jpeg *j, int n)
02152 {
02153    unsigned int k;
02154    int sgn;
02155    if (j->code_bits < n) stbi__grow_buffer_unsafe(j);
02156    if (j->code_bits < n) return 0; // ran out of bits from stream, return 0s intead of continuing
```

```
02157
02158    sgn = j->code_buffer >> 31; // sign bit always in MSB; 0 if MSB clear (positive), 1 if MSB set
     (negative)
02159    k = stbi_lrot(j->code_buffer, n);
02160    j->code_buffer = k & ~stbi__bmask[n];
02161    k &= stbi__bmask[n];
02162    j->code_bits -= n;
02163    return k + (stbi__jbias[n] & (sgn - 1));
02164 }
02165
02166 // get some unsigned bits
02167 stbi_inline static int stbi__jpeg_get_bits(stbi__jpeg *j, int n)
02168 {
02169    unsigned int k;
02170    if (j->code_bits < n) stbi__grow_buffer_unsafe(j);
02171    if (j->code_bits < n) return 0; // ran out of bits from stream, return 0s intead of continuing
02172    k = stbi_lrot(j->code_buffer, n);
02173    j->code_buffer = k & ~stbi__bmask[n];
02174    k &= stbi__bmask[n];
02175    j->code_bits -= n;
02176    return k;
02177 }
02178
02179 stbi_inline static int stbi__jpeg_get_bit(stbi__jpeg *j)
02180 {
02181    unsigned int k;
02182    if (j->code_bits < 1) stbi__grow_buffer_unsafe(j);
02183    if (j->code_bits < 1) return 0; // ran out of bits from stream, return 0s intead of continuing
02184    k = j->code_buffer;
02185    j->code_buffer <<= 1;
02186    --j->code_bits;
02187    return k & 0x80000000;
02188 }
02189
02190 // given a value that's at position X in the zigzag stream,
02191 // where does it appear in the 8x8 matrix coded as row-major?
02192 static const stbi_uc stbi__jpeg_dezigzag[64+15] =
02193 {
02194     0,  1,  8, 16,  9,  2,  3, 10,
02195    17, 24, 32, 25, 18, 11,  4,  5,
02196    12, 19, 26, 33, 40, 48, 41, 34,
02197    27, 20, 13,  6,  7, 14, 21, 28,
02198    35, 42, 49, 56, 57, 50, 43, 36,
02199    29, 22, 15, 23, 30, 37, 44, 51,
02200    58, 59, 52, 45, 38, 31, 39, 46,
02201    53, 60, 61, 54, 47, 55, 62, 63,
02202    // let corrupt input sample past end
02203    63, 63, 63, 63, 63, 63, 63, 63,
02204    63, 63, 63, 63, 63, 63, 63
02205 };
02206
02207 // decode one 64-entry block--
02208 static int stbi__jpeg_decode_block(stbi__jpeg *j, short data[64], stbi__huffman *hdc, stbi__huffman
     *hac, stbi__int16 *fac, int b, stbi__uint16 *dequant)
02209 {
02210    int diff,dc,k;
02211    int t;
02212
02213    if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
02214    t = stbi__jpeg_huff_decode(j, hdc);
02215    if (t < 0 || t > 15) return stbi__err("bad huffman code","Corrupt JPEG");
02216
02217    // 0 all the ac values now so we can do it 32-bits at a time
02218    memset(data,0,64*sizeof(data[0]));
02219
02220    diff = t ? stbi__extend_receive(j, t) : 0;
02221    if (!stbi__addints_valid(j->img_comp[b].dc_pred, diff)) return stbi__err("bad delta","Corrupt
     JPEG");
02222    dc = j->img_comp[b].dc_pred + diff;
02223    j->img_comp[b].dc_pred = dc;
02224    if (!stbi__mul2shorts_valid(dc, dequant[0])) return stbi__err("can't merge dc and ac", "Corrupt
     JPEG");
02225    data[0] = (short) (dc * dequant[0]);
02226
02227    // decode AC components, see JPEG spec
02228    k = 1;
02229    do {
02230       unsigned int zig;
02231       int c,r,s;
02232       if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
02233       c = (j->code_buffer >> (32 - FAST_BITS)) & ((1 << FAST_BITS)-1);
02234       r = fac[c];
02235       if (r) { // fast-AC path
02236          k += (r >> 4) & 15; // run
02237          s = r & 15; // combined length
02238          if (s > j->code_bits) return stbi__err("bad huffman code", "Combined length longer than code
     bits available");
```

```
02239          j->code_buffer <<= s;
02240          j->code_bits -= s;
02241          // decode into unzigzag'd location
02242          zig = stbi__jpeg_dezigzag[k++];
02243          data[zig] = (short) ((r >> 8) * dequant[zig]);
02244       } else {
02245          int rs = stbi__jpeg_huff_decode(j, hac);
02246          if (rs < 0) return stbi__err("bad huffman code","Corrupt JPEG");
02247          s = rs & 15;
02248          r = rs >> 4;
02249          if (s == 0) {
02250             if (rs != 0xf0) break; // end block
02251             k += 16;
02252          } else {
02253             k += r;
02254             // decode into unzigzag'd location
02255             zig = stbi__jpeg_dezigzag[k++];
02256             data[zig] = (short) (stbi__extend_receive(j,s) * dequant[zig]);
02257          }
02258       }
02259    } while (k < 64);
02260    return 1;
02261 }
02262
02263 static int stbi__jpeg_decode_block_prog_dc(stbi__jpeg *j, short data[64], stbi__huffman *hdc, int b)
02264 {
02265    int diff,dc;
02266    int t;
02267    if (j->spec_end != 0) return stbi__err("can't merge dc and ac", "Corrupt JPEG");
02268
02269    if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
02270
02271    if (j->succ_high == 0) {
02272       // first scan for DC coefficient, must be first
02273       memset(data,0,64*sizeof(data[0])); // 0 all the ac values now
02274       t = stbi__jpeg_huff_decode(j, hdc);
02275       if (t < 0 || t > 15) return stbi__err("can't merge dc and ac", "Corrupt JPEG");
02276       diff = t ? stbi__extend_receive(j, t) : 0;
02277
02278       if (!stbi__addints_valid(j->img_comp[b].dc_pred, diff)) return stbi__err("bad delta", "Corrupt
      JPEG");
02279       dc = j->img_comp[b].dc_pred + diff;
02280       j->img_comp[b].dc_pred = dc;
02281       if (!stbi__mul2shorts_valid(dc, 1 << j->succ_low)) return stbi__err("can't merge dc and ac",
      "Corrupt JPEG");
02282       data[0] = (short) (dc * (1 << j->succ_low));
02283    } else {
02284       // refinement scan for DC coefficient
02285       if (stbi__jpeg_get_bit(j))
02286          data[0] += (short) (1 << j->succ_low);
02287    }
02288    return 1;
02289 }
02290
02291 // @OPTIMIZE: store non-zigzagged during the decode passes,
02292 // and only de-zigzag when dequantizing
02293 static int stbi__jpeg_decode_block_prog_ac(stbi__jpeg *j, short data[64], stbi__huffman *hac,
      stbi__int16 *fac)
02294 {
02295    int k;
02296    if (j->spec_start == 0) return stbi__err("can't merge dc and ac", "Corrupt JPEG");
02297
02298    if (j->succ_high == 0) {
02299       int shift = j->succ_low;
02300
02301       if (j->eob_run) {
02302          --j->eob_run;
02303          return 1;
02304       }
02305
02306       k = j->spec_start;
02307       do {
02308          unsigned int zig;
02309          int c,r,s;
02310          if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
02311          c = (j->code_buffer >> (32 - FAST_BITS)) & ((1 << FAST_BITS)-1);
02312          r = fac[c];
02313          if (r) { // fast-AC path
02314             k += (r >> 4) & 15; // run
02315             s = r & 15; // combined length
02316             if (s > j->code_bits) return stbi__err("bad huffman code", "Combined length longer than
      code bits available");
02317             j->code_buffer <<= s;
02318             j->code_bits -= s;
02319             zig = stbi__jpeg_dezigzag[k++];
02320             data[zig] = (short) ((r >> 8) * (1 << shift));
02321          } else {
```

```
02322            int rs = stbi__jpeg_huff_decode(j, hac);
02323            if (rs < 0) return stbi__err("bad huffman code","Corrupt JPEG");
02324            s = rs & 15;
02325            r = rs » 4;
02326            if (s == 0) {
02327               if (r < 15) {
02328                  j->eob_run = (1 « r);
02329                  if (r)
02330                     j->eob_run += stbi__jpeg_get_bits(j, r);
02331                  --j->eob_run;
02332                  break;
02333               }
02334               k += 16;
02335            } else {
02336               k += r;
02337               zig = stbi__jpeg_dezigzag[k++];
02338               data[zig] = (short) (stbi__extend_receive(j,s) * (1 « shift));
02339            }
02340         }
02341      } while (k <= j->spec_end);
02342   } else {
02343      // refinement scan for these AC coefficients
02344
02345      short bit = (short) (1 « j->succ_low);
02346
02347      if (j->eob_run) {
02348         --j->eob_run;
02349         for (k = j->spec_start; k <= j->spec_end; ++k) {
02350            short *p = &data[stbi__jpeg_dezigzag[k]];
02351            if (*p != 0)
02352               if (stbi__jpeg_get_bit(j))
02353                  if ((*p & bit)==0) {
02354                     if (*p > 0)
02355                        *p += bit;
02356                     else
02357                        *p -= bit;
02358                  }
02359         }
02360      } else {
02361         k = j->spec_start;
02362         do {
02363            int r,s;
02364            int rs = stbi__jpeg_huff_decode(j, hac); // @OPTIMIZE see if we can use the fast path
   here, advance-by-r is so slow, eh
02365            if (rs < 0) return stbi__err("bad huffman code","Corrupt JPEG");
02366            s = rs & 15;
02367            r = rs » 4;
02368            if (s == 0) {
02369               if (r < 15) {
02370                  j->eob_run = (1 « r) - 1;
02371                  if (r)
02372                     j->eob_run += stbi__jpeg_get_bits(j, r);
02373                  r = 64; // force end of block
02374               } else {
02375                  // r=15 s=0 should write 16 0s, so we just do
02376                  // a run of 15 0s and then write s (which is 0),
02377                  // so we don't have to do anything special here
02378               }
02379            } else {
02380               if (s != 1) return stbi__err("bad huffman code", "Corrupt JPEG");
02381               // sign bit
02382               if (stbi__jpeg_get_bit(j))
02383                  s = bit;
02384               else
02385                  s = -bit;
02386            }
02387
02388            // advance by r
02389            while (k <= j->spec_end) {
02390               short *p = &data[stbi__jpeg_dezigzag[k++]];
02391               if (*p != 0) {
02392                  if (stbi__jpeg_get_bit(j))
02393                     if ((*p & bit)==0) {
02394                        if (*p > 0)
02395                           *p += bit;
02396                        else
02397                           *p -= bit;
02398                     }
02399               } else {
02400                  if (r == 0) {
02401                     *p = (short) s;
02402                     break;
02403                  }
02404                  --r;
02405               }
02406            }
02407         } while (k <= j->spec_end);
```

```
02408       }
02409    }
02410    return 1;
02411 }
02412
02413 // take a -128..127 value and stbi__clamp it and convert to 0..255
02414 stbi_inline static stbi_uc stbi__clamp(int x)
02415 {
02416    // trick to use a single test to catch both cases
02417    if ((unsigned int) x > 255) {
02418       if (x < 0) return 0;
02419       if (x > 255) return 255;
02420    }
02421    return (stbi_uc) x;
02422 }
02423
02424 #define stbi__f2f(x)  ((int) (((x) * 4096 + 0.5)))
02425 #define stbi__fsh(x)  ((x) * 4096)
02426
02427 // derived from jidctint -- DCT_ISLOW
02428 #define STBI__IDCT_1D(s0,s1,s2,s3,s4,s5,s6,s7) \
02429    int t0,t1,t2,t3,p1,p2,p3,p4,p5,x0,x1,x2,x3; \
02430    p2 = s2;                                    \
02431    p3 = s6;                                    \
02432    p1 = (p2+p3) * stbi__f2f(0.5411961f);       \
02433    t2 = p1 + p3*stbi__f2f(-1.847759065f);      \
02434    t3 = p1 + p2*stbi__f2f( 0.765366865f);      \
02435    p2 = s0;                                    \
02436    p3 = s4;                                    \
02437    t0 = stbi__fsh(p2+p3);                      \
02438    t1 = stbi__fsh(p2-p3);                      \
02439    x0 = t0+t3;                                 \
02440    x3 = t0-t3;                                 \
02441    x1 = t1+t2;                                 \
02442    x2 = t1-t2;                                 \
02443    t0 = s7;                                    \
02444    t1 = s5;                                    \
02445    t2 = s3;                                    \
02446    t3 = s1;                                    \
02447    p3 = t0+t2;                                 \
02448    p4 = t1+t3;                                 \
02449    p1 = t0+t3;                                 \
02450    p2 = t1+t2;                                 \
02451    p5 = (p3+p4)*stbi__f2f( 1.175875602f);      \
02452    t0 = t0*stbi__f2f( 0.298631336f);           \
02453    t1 = t1*stbi__f2f( 2.053119869f);           \
02454    t2 = t2*stbi__f2f( 3.072711026f);           \
02455    t3 = t3*stbi__f2f( 1.501321110f);           \
02456    p1 = p5 + p1*stbi__f2f(-0.899976223f);      \
02457    p2 = p5 + p2*stbi__f2f(-2.562915447f);      \
02458    p3 = p3*stbi__f2f(-1.961570560f);           \
02459    p4 = p4*stbi__f2f(-0.390180644f);           \
02460    t3 += p1+p4;                                \
02461    t2 += p2+p3;                                \
02462    t1 += p2+p4;                                \
02463    t0 += p1+p3;
02464
02465 static void stbi__idct_block(stbi_uc *out, int out_stride, short data[64])
02466 {
02467    int i,val[64],*v=val;
02468    stbi_uc *o;
02469    short *d = data;
02470
02471    // columns
02472    for (i=0; i < 8; ++i,++d, ++v) {
02473       // if all zeroes, shortcut -- this avoids dequantizing 0s and IDCTing
02474       if (d[ 8]==0 && d[16]==0 && d[24]==0 && d[32]==0
02475            && d[40]==0 && d[48]==0 && d[56]==0) {
02476          //    no shortcut                 0     seconds
02477          //    (1|2|3|4|5|6|7)==0          0     seconds
02478          //    all separate               -0.047 seconds
02479          //    1 && 2|3 && 4|5 && 6|7:    -0.047 seconds
02480          int dcterm = d[0]*4;
02481          v[0] = v[8] = v[16] = v[24] = v[32] = v[40] = v[48] = v[56] = dcterm;
02482       } else {
02483          STBI__IDCT_1D(d[ 0],d[ 8],d[16],d[24],d[32],d[40],d[48],d[56])
02484          // constants scaled things up by 1<<12; let's bring them back
02485          // down, but keep 2 extra bits of precision
02486          x0 += 512; x1 += 512; x2 += 512; x3 += 512;
02487          v[ 0] = (x0+t3) >> 10;
02488          v[56] = (x0-t3) >> 10;
02489          v[ 8] = (x1+t2) >> 10;
02490          v[48] = (x1-t2) >> 10;
02491          v[16] = (x2+t1) >> 10;
02492          v[40] = (x2-t1) >> 10;
02493          v[24] = (x3+t0) >> 10;
02494          v[32] = (x3-t0) >> 10;
```

```
02495       }
02496    }
02497
02498    for (i=0, v=val, o=out; i < 8; ++i,v+=8,o+=out_stride) {
02499       // no fast case since the first 1D IDCT spread components out
02500       STBI__IDCT_1D(v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7])
02501       // constants scaled things up by 1«12, plus we had 1«2 from first
02502       // loop, plus horizontal and vertical each scale by sqrt(8) so together
02503       // we've got an extra 1«3, so 1«17 total we need to remove.
02504       // so we want to round that, which means adding 0.5 * 1«17,
02505       // aka 65536. Also, we'll end up with -128 to 127 that we want
02506       // to encode as 0..255 by adding 128, so we'll add that before the shift
02507       x0 += 65536 + (128«17);
02508       x1 += 65536 + (128«17);
02509       x2 += 65536 + (128«17);
02510       x3 += 65536 + (128«17);
02511       // tried computing the shifts into temps, or'ing the temps to see
02512       // if any were out of range, but that was slower
02513       o[0] = stbi__clamp((x0+t3) » 17);
02514       o[7] = stbi__clamp((x0-t3) » 17);
02515       o[1] = stbi__clamp((x1+t2) » 17);
02516       o[6] = stbi__clamp((x1-t2) » 17);
02517       o[2] = stbi__clamp((x2+t1) » 17);
02518       o[5] = stbi__clamp((x2-t1) » 17);
02519       o[3] = stbi__clamp((x3+t0) » 17);
02520       o[4] = stbi__clamp((x3-t0) » 17);
02521    }
02522 }
02523
02524 #ifdef STBI_SSE2
02525 // sse2 integer IDCT. not the fastest possible implementation but it
02526 // produces bit-identical results to the generic C version so it's
02527 // fully "transparent".
02528 static void stbi__idct_simd(stbi_uc *out, int out_stride, short data[64])
02529 {
02530    // This is constructed to match our regular (generic) integer IDCT exactly.
02531    __m128i row0, row1, row2, row3, row4, row5, row6, row7;
02532    __m128i tmp;
02533
02534    // dot product constant: even elems=x, odd elems=y
02535    #define dct_const(x,y)  _mm_setr_epi16((x),(y),(x),(y),(x),(y),(x),(y))
02536
02537    // out(0) = c0[even]*x + c0[odd]*y   (c0, x, y 16-bit, out 32-bit)
02538    // out(1) = c1[even]*x + c1[odd]*y
02539    #define dct_rot(out0,out1, x,y,c0,c1) \
02540       __m128i c0##lo = _mm_unpacklo_epi16((x),(y)); \
02541       __m128i c0##hi = _mm_unpackhi_epi16((x),(y)); \
02542       __m128i out0##_l = _mm_madd_epi16(c0##lo, c0); \
02543       __m128i out0##_h = _mm_madd_epi16(c0##hi, c0); \
02544       __m128i out1##_l = _mm_madd_epi16(c0##lo, c1); \
02545       __m128i out1##_h = _mm_madd_epi16(c0##hi, c1)
02546
02547    // out = in « 12  (in 16-bit, out 32-bit)
02548    #define dct_widen(out, in) \
02549       __m128i out##_l = _mm_srai_epi32(_mm_unpacklo_epi16(_mm_setzero_si128(), (in)), 4); \
02550       __m128i out##_h = _mm_srai_epi32(_mm_unpackhi_epi16(_mm_setzero_si128(), (in)), 4)
02551
02552    // wide add
02553    #define dct_wadd(out, a, b) \
02554       __m128i out##_l = _mm_add_epi32(a##_l, b##_l); \
02555       __m128i out##_h = _mm_add_epi32(a##_h, b##_h)
02556
02557    // wide sub
02558    #define dct_wsub(out, a, b) \
02559       __m128i out##_l = _mm_sub_epi32(a##_l, b##_l); \
02560       __m128i out##_h = _mm_sub_epi32(a##_h, b##_h)
02561
02562    // butterfly a/b, add bias, then shift by "s" and pack
02563    #define dct_bfly32o(out0, out1, a,b,bias,s) \
02564       { \
02565          __m128i abiased_l = _mm_add_epi32(a##_l, bias); \
02566          __m128i abiased_h = _mm_add_epi32(a##_h, bias); \
02567          dct_wadd(sum, abiased, b); \
02568          dct_wsub(dif, abiased, b); \
02569          out0 = _mm_packs_epi32(_mm_srai_epi32(sum_l, s), _mm_srai_epi32(sum_h, s)); \
02570          out1 = _mm_packs_epi32(_mm_srai_epi32(dif_l, s), _mm_srai_epi32(dif_h, s)); \
02571       }
02572
02573    // 8-bit interleave step (for transposes)
02574    #define dct_interleave8(a, b) \
02575       tmp = a; \
02576       a = _mm_unpacklo_epi8(a, b); \
02577       b = _mm_unpackhi_epi8(tmp, b)
02578
02579    // 16-bit interleave step (for transposes)
02580    #define dct_interleave16(a, b) \
02581       tmp = a; \
```

```
02582       a = _mm_unpacklo_epi16(a, b); \
02583       b = _mm_unpackhi_epi16(tmp, b)
02584
02585    #define dct_pass(bias,shift) \
02586       { \
02587          /* even part */ \
02588          dct_rot(t2e,t3e, row2,row6, rot0_0,rot0_1); \
02589          __m128i sum04 = _mm_add_epi16(row0, row4); \
02590          __m128i dif04 = _mm_sub_epi16(row0, row4); \
02591          dct_widen(t0e, sum04); \
02592          dct_widen(t1e, dif04); \
02593          dct_wadd(x0, t0e, t3e); \
02594          dct_wsub(x3, t0e, t3e); \
02595          dct_wadd(x1, t1e, t2e); \
02596          dct_wsub(x2, t1e, t2e); \
02597          /* odd part */ \
02598          dct_rot(y0o,y2o, row7,row3, rot2_0,rot2_1); \
02599          dct_rot(y1o,y3o, row5,row1, rot3_0,rot3_1); \
02600          __m128i sum17 = _mm_add_epi16(row1, row7); \
02601          __m128i sum35 = _mm_add_epi16(row3, row5); \
02602          dct_rot(y4o,y5o, sum17,sum35, rot1_0,rot1_1); \
02603          dct_wadd(x4, y0o, y4o); \
02604          dct_wadd(x5, y1o, y5o); \
02605          dct_wadd(x6, y2o, y5o); \
02606          dct_wadd(x7, y3o, y4o); \
02607          dct_bfly32o(row0,row7, x0,x7,bias,shift); \
02608          dct_bfly32o(row1,row6, x1,x6,bias,shift); \
02609          dct_bfly32o(row2,row5, x2,x5,bias,shift); \
02610          dct_bfly32o(row3,row4, x3,x4,bias,shift); \
02611       }
02612
02613    __m128i rot0_0 = dct_const(stbi__f2f(0.5411961f), stbi__f2f(0.5411961f) +
       stbi__f2f(-1.847759065f));
02614    __m128i rot0_1 = dct_const(stbi__f2f(0.5411961f) + stbi__f2f( 0.765366865f),
       stbi__f2f(0.5411961f));
02615    __m128i rot1_0 = dct_const(stbi__f2f(1.175875602f) + stbi__f2f(-0.899976223f),
       stbi__f2f(1.175875602f));
02616    __m128i rot1_1 = dct_const(stbi__f2f(1.175875602f), stbi__f2f(1.175875602f) +
       stbi__f2f(-2.562915447f));
02617    __m128i rot2_0 = dct_const(stbi__f2f(-1.961570560f) + stbi__f2f( 0.298631336f),
       stbi__f2f(-1.961570560f));
02618    __m128i rot2_1 = dct_const(stbi__f2f(-1.961570560f), stbi__f2f(-1.961570560f) + stbi__f2f(
       3.072711026f));
02619    __m128i rot3_0 = dct_const(stbi__f2f(-0.390180644f) + stbi__f2f( 2.053119869f),
       stbi__f2f(-0.390180644f));
02620    __m128i rot3_1 = dct_const(stbi__f2f(-0.390180644f), stbi__f2f(-0.390180644f) + stbi__f2f(
       1.501321110f));
02621
02622    // rounding biases in column/row passes, see stbi__idct_block for explanation.
02623    __m128i bias_0 = _mm_set1_epi32(512);
02624    __m128i bias_1 = _mm_set1_epi32(65536 + (128<<17));
02625
02626    // load
02627    row0 = _mm_load_si128((const __m128i *) (data + 0*8));
02628    row1 = _mm_load_si128((const __m128i *) (data + 1*8));
02629    row2 = _mm_load_si128((const __m128i *) (data + 2*8));
02630    row3 = _mm_load_si128((const __m128i *) (data + 3*8));
02631    row4 = _mm_load_si128((const __m128i *) (data + 4*8));
02632    row5 = _mm_load_si128((const __m128i *) (data + 5*8));
02633    row6 = _mm_load_si128((const __m128i *) (data + 6*8));
02634    row7 = _mm_load_si128((const __m128i *) (data + 7*8));
02635
02636    // column pass
02637    dct_pass(bias_0, 10);
02638
02639    {
02640       // 16bit 8x8 transpose pass 1
02641       dct_interleave16(row0, row4);
02642       dct_interleave16(row1, row5);
02643       dct_interleave16(row2, row6);
02644       dct_interleave16(row3, row7);
02645
02646       // transpose pass 2
02647       dct_interleave16(row0, row2);
02648       dct_interleave16(row1, row3);
02649       dct_interleave16(row4, row6);
02650       dct_interleave16(row5, row7);
02651
02652       // transpose pass 3
02653       dct_interleave16(row0, row1);
02654       dct_interleave16(row2, row3);
02655       dct_interleave16(row4, row5);
02656       dct_interleave16(row6, row7);
02657    }
02658
02659    // row pass
02660    dct_pass(bias_1, 17);
```

```
02661
02662    {
02663       // pack
02664       __m128i p0 = _mm_packus_epi16(row0, row1); // a0a1a2a3...a7b0b1b2b3...b7
02665       __m128i p1 = _mm_packus_epi16(row2, row3);
02666       __m128i p2 = _mm_packus_epi16(row4, row5);
02667       __m128i p3 = _mm_packus_epi16(row6, row7);
02668
02669       // 8bit 8x8 transpose pass 1
02670       dct_interleave8(p0, p2); // a0e0a1e1...
02671       dct_interleave8(p1, p3); // c0g0c1g1...
02672
02673       // transpose pass 2
02674       dct_interleave8(p0, p1); // a0c0e0g0...
02675       dct_interleave8(p2, p3); // b0d0f0h0...
02676
02677       // transpose pass 3
02678       dct_interleave8(p0, p2); // a0b0c0d0...
02679       dct_interleave8(p1, p3); // a4b4c4d4...
02680
02681       // store
02682       _mm_storel_epi64((__m128i *) out, p0); out += out_stride;
02683       _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p0, 0x4e)); out += out_stride;
02684       _mm_storel_epi64((__m128i *) out, p2); out += out_stride;
02685       _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p2, 0x4e)); out += out_stride;
02686       _mm_storel_epi64((__m128i *) out, p1); out += out_stride;
02687       _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p1, 0x4e)); out += out_stride;
02688       _mm_storel_epi64((__m128i *) out, p3); out += out_stride;
02689       _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p3, 0x4e));
02690    }
02691
02692 #undef dct_const
02693 #undef dct_rot
02694 #undef dct_widen
02695 #undef dct_wadd
02696 #undef dct_wsub
02697 #undef dct_bfly32o
02698 #undef dct_interleave8
02699 #undef dct_interleave16
02700 #undef dct_pass
02701 }
02702
02703 #endif // STBI_SSE2
02704
02705 #ifdef STBI_NEON
02706
02707 // NEON integer IDCT. should produce bit-identical
02708 // results to the generic C version.
02709 static void stbi__idct_simd(stbi_uc *out, int out_stride, short data[64])
02710 {
02711    int16x8_t row0, row1, row2, row3, row4, row5, row6, row7;
02712
02713    int16x4_t rot0_0 = vdup_n_s16(stbi__f2f(0.5411961f));
02714    int16x4_t rot0_1 = vdup_n_s16(stbi__f2f(-1.847759065f));
02715    int16x4_t rot0_2 = vdup_n_s16(stbi__f2f( 0.765366865f));
02716    int16x4_t rot1_0 = vdup_n_s16(stbi__f2f( 1.175875602f));
02717    int16x4_t rot1_1 = vdup_n_s16(stbi__f2f(-0.899976223f));
02718    int16x4_t rot1_2 = vdup_n_s16(stbi__f2f(-2.562915447f));
02719    int16x4_t rot2_0 = vdup_n_s16(stbi__f2f(-1.961570560f));
02720    int16x4_t rot2_1 = vdup_n_s16(stbi__f2f(-0.390180644f));
02721    int16x4_t rot3_0 = vdup_n_s16(stbi__f2f( 0.298631336f));
02722    int16x4_t rot3_1 = vdup_n_s16(stbi__f2f( 2.053119869f));
02723    int16x4_t rot3_2 = vdup_n_s16(stbi__f2f( 3.072711026f));
02724    int16x4_t rot3_3 = vdup_n_s16(stbi__f2f( 1.501321110f));
02725
02726 #define dct_long_mul(out, inq, coeff) \
02727    int32x4_t out##_l = vmull_s16(vget_low_s16(inq), coeff); \
02728    int32x4_t out##_h = vmull_s16(vget_high_s16(inq), coeff)
02729
02730 #define dct_long_mac(out, acc, inq, coeff) \
02731    int32x4_t out##_l = vmlal_s16(acc##_l, vget_low_s16(inq), coeff); \
02732    int32x4_t out##_h = vmlal_s16(acc##_h, vget_high_s16(inq), coeff)
02733
02734 #define dct_widen(out, inq) \
02735    int32x4_t out##_l = vshll_n_s16(vget_low_s16(inq), 12); \
02736    int32x4_t out##_h = vshll_n_s16(vget_high_s16(inq), 12)
02737
02738 // wide add
02739 #define dct_wadd(out, a, b) \
02740    int32x4_t out##_l = vaddq_s32(a##_l, b##_l); \
02741    int32x4_t out##_h = vaddq_s32(a##_h, b##_h)
02742
02743 // wide sub
02744 #define dct_wsub(out, a, b) \
02745    int32x4_t out##_l = vsubq_s32(a##_l, b##_l); \
02746    int32x4_t out##_h = vsubq_s32(a##_h, b##_h)
02747
```

```
02748 // butterfly a/b, then shift using "shiftop" by "s" and pack
02749 #define dct_bfly32o(out0,out1, a,b,shiftop,s) \
02750    { \
02751       dct_wadd(sum, a, b); \
02752       dct_wsub(dif, a, b); \
02753       out0 = vcombine_s16(shiftop(sum_l, s), shiftop(sum_h, s)); \
02754       out1 = vcombine_s16(shiftop(dif_l, s), shiftop(dif_h, s)); \
02755    }
02756
02757 #define dct_pass(shiftop, shift) \
02758    { \
02759       /* even part */ \
02760       int16x8_t sum26 = vaddq_s16(row2, row6); \
02761       dct_long_mul(p1e, sum26, rot0_0); \
02762       dct_long_mac(t2e, p1e, row6, rot0_1); \
02763       dct_long_mac(t3e, p1e, row2, rot0_2); \
02764       int16x8_t sum04 = vaddq_s16(row0, row4); \
02765       int16x8_t dif04 = vsubq_s16(row0, row4); \
02766       dct_widen(t0e, sum04); \
02767       dct_widen(t1e, dif04); \
02768       dct_wadd(x0, t0e, t3e); \
02769       dct_wsub(x3, t0e, t3e); \
02770       dct_wadd(x1, t1e, t2e); \
02771       dct_wsub(x2, t1e, t2e); \
02772       /* odd part */ \
02773       int16x8_t sum15 = vaddq_s16(row1, row5); \
02774       int16x8_t sum17 = vaddq_s16(row1, row7); \
02775       int16x8_t sum35 = vaddq_s16(row3, row5); \
02776       int16x8_t sum37 = vaddq_s16(row3, row7); \
02777       int16x8_t sumodd = vaddq_s16(sum17, sum35); \
02778       dct_long_mul(p5o, sumodd, rot1_0); \
02779       dct_long_mac(p1o, p5o, sum17, rot1_1); \
02780       dct_long_mac(p2o, p5o, sum35, rot1_2); \
02781       dct_long_mul(p3o, sum37, rot2_0); \
02782       dct_long_mul(p4o, sum15, rot2_1); \
02783       dct_wadd(sump13o, p1o, p3o); \
02784       dct_wadd(sump24o, p2o, p4o); \
02785       dct_wadd(sump23o, p2o, p3o); \
02786       dct_wadd(sump14o, p1o, p4o); \
02787       dct_long_mac(x4, sump13o, row7, rot3_0); \
02788       dct_long_mac(x5, sump24o, row5, rot3_1); \
02789       dct_long_mac(x6, sump23o, row3, rot3_2); \
02790       dct_long_mac(x7, sump14o, row1, rot3_3); \
02791       dct_bfly32o(row0,row7, x0,x7,shiftop,shift); \
02792       dct_bfly32o(row1,row6, x1,x6,shiftop,shift); \
02793       dct_bfly32o(row2,row5, x2,x5,shiftop,shift); \
02794       dct_bfly32o(row3,row4, x3,x4,shiftop,shift); \
02795    }
02796
02797    // load
02798    row0 = vld1q_s16(data + 0*8);
02799    row1 = vld1q_s16(data + 1*8);
02800    row2 = vld1q_s16(data + 2*8);
02801    row3 = vld1q_s16(data + 3*8);
02802    row4 = vld1q_s16(data + 4*8);
02803    row5 = vld1q_s16(data + 5*8);
02804    row6 = vld1q_s16(data + 6*8);
02805    row7 = vld1q_s16(data + 7*8);
02806
02807    // add DC bias
02808    row0 = vaddq_s16(row0, vsetq_lane_s16(1024, vdupq_n_s16(0), 0));
02809
02810    // column pass
02811    dct_pass(vrshrn_n_s32, 10);
02812
02813    // 16bit 8x8 transpose
02814    {
02815 // these three map to a single VTRN.16, VTRN.32, and VSWP, respectively.
02816 // whether compilers actually get this is another story, sadly.
02817 #define dct_trn16(x, y) { int16x8x2_t t = vtrnq_s16(x, y); x = t.val[0]; y = t.val[1]; }
02818 #define dct_trn32(x, y) { int32x4x2_t t = vtrnq_s32(vreinterpretq_s32_s16(x),
      vreinterpretq_s32_s16(y)); x = vreinterpretq_s16_s32(t.val[0]); y = vreinterpretq_s16_s32(t.val[1]); }
02819 #define dct_trn64(x, y) { int16x8_t x0 = x; int16x8_t y0 = y; x = vcombine_s16(vget_low_s16(x0),
      vget_low_s16(y0)); y = vcombine_s16(vget_high_s16(x0), vget_high_s16(y0)); }
02820
02821       // pass 1
02822       dct_trn16(row0, row1); // a0b0a2b2a4b4a6b6
02823       dct_trn16(row2, row3);
02824       dct_trn16(row4, row5);
02825       dct_trn16(row6, row7);
02826
02827       // pass 2
02828       dct_trn32(row0, row2); // a0b0c0d0a4b4c4d4
02829       dct_trn32(row1, row3);
02830       dct_trn32(row4, row6);
02831       dct_trn32(row5, row7);
02832
```

```
02833       // pass 3
02834       dct_trn64(row0, row4); // a0b0c0d0e0f0g0h0
02835       dct_trn64(row1, row5);
02836       dct_trn64(row2, row6);
02837       dct_trn64(row3, row7);
02838
02839 #undef dct_trn16
02840 #undef dct_trn32
02841 #undef dct_trn64
02842    }
02843
02844    // row pass
02845    // vrshrn_n_s32 only supports shifts up to 16, we need
02846    // 17. so do a non-rounding shift of 16 first then follow
02847    // up with a rounding shift by 1.
02848    dct_pass(vshrn_n_s32, 16);
02849
02850    {
02851       // pack and round
02852       uint8x8_t p0 = vqrshrun_n_s16(row0, 1);
02853       uint8x8_t p1 = vqrshrun_n_s16(row1, 1);
02854       uint8x8_t p2 = vqrshrun_n_s16(row2, 1);
02855       uint8x8_t p3 = vqrshrun_n_s16(row3, 1);
02856       uint8x8_t p4 = vqrshrun_n_s16(row4, 1);
02857       uint8x8_t p5 = vqrshrun_n_s16(row5, 1);
02858       uint8x8_t p6 = vqrshrun_n_s16(row6, 1);
02859       uint8x8_t p7 = vqrshrun_n_s16(row7, 1);
02860
02861       // again, these can translate into one instruction, but often don't.
02862 #define dct_trn8_8(x, y) { uint8x8x2_t t = vtrn_u8(x, y); x = t.val[0]; y = t.val[1]; }
02863 #define dct_trn8_16(x, y) { uint16x4x2_t t = vtrn_u16(vreinterpret_u16_u8(x), vreinterpret_u16_u8(y));
      x = vreinterpret_u8_u16(t.val[0]); y = vreinterpret_u8_u16(t.val[1]); }
02864 #define dct_trn8_32(x, y) { uint32x2x2_t t = vtrn_u32(vreinterpret_u32_u8(x), vreinterpret_u32_u8(y));
      x = vreinterpret_u8_u32(t.val[0]); y = vreinterpret_u8_u32(t.val[1]); }
02865
02866       // sadly can't use interleaved stores here since we only write
02867       // 8 bytes to each scan line!
02868
02869       // 8x8 8-bit transpose pass 1
02870       dct_trn8_8(p0, p1);
02871       dct_trn8_8(p2, p3);
02872       dct_trn8_8(p4, p5);
02873       dct_trn8_8(p6, p7);
02874
02875       // pass 2
02876       dct_trn8_16(p0, p2);
02877       dct_trn8_16(p1, p3);
02878       dct_trn8_16(p4, p6);
02879       dct_trn8_16(p5, p7);
02880
02881       // pass 3
02882       dct_trn8_32(p0, p4);
02883       dct_trn8_32(p1, p5);
02884       dct_trn8_32(p2, p6);
02885       dct_trn8_32(p3, p7);
02886
02887       // store
02888       vst1_u8(out, p0); out += out_stride;
02889       vst1_u8(out, p1); out += out_stride;
02890       vst1_u8(out, p2); out += out_stride;
02891       vst1_u8(out, p3); out += out_stride;
02892       vst1_u8(out, p4); out += out_stride;
02893       vst1_u8(out, p5); out += out_stride;
02894       vst1_u8(out, p6); out += out_stride;
02895       vst1_u8(out, p7);
02896
02897 #undef dct_trn8_8
02898 #undef dct_trn8_16
02899 #undef dct_trn8_32
02900    }
02901
02902 #undef dct_long_mul
02903 #undef dct_long_mac
02904 #undef dct_widen
02905 #undef dct_wadd
02906 #undef dct_wsub
02907 #undef dct_bfly32o
02908 #undef dct_pass
02909 }
02910
02911 #endif // STBI_NEON
02912
02913 #define STBI__MARKER_none  0xff
02914 // if there's a pending marker from the entropy stream, return that
02915 // otherwise, fetch from the stream and get a marker. if there's no
02916 // marker, return 0xff, which is never a valid marker value
02917 static stbi_uc stbi__get_marker(stbi__jpeg *j)
```

```
02918 {
02919    stbi_uc x;
02920    if (j->marker != STBI__MARKER_none) { x = j->marker; j->marker = STBI__MARKER_none; return x; }
02921    x = stbi__get8(j->s);
02922    if (x != 0xff) return STBI__MARKER_none;
02923    while (x == 0xff)
02924       x = stbi__get8(j->s); // consume repeated 0xff fill bytes
02925    return x;
02926 }
02927
02928 // in each scan, we'll have scan_n components, and the order
02929 // of the components is specified by order[]
02930 #define STBI__RESTART(x)     ((x) >= 0xd0 && (x) <= 0xd7)
02931
02932 // after a restart interval, stbi__jpeg_reset the entropy decoder and
02933 // the dc prediction
02934 static void stbi__jpeg_reset(stbi__jpeg *j)
02935 {
02936    j->code_bits = 0;
02937    j->code_buffer = 0;
02938    j->nomore = 0;
02939    j->img_comp[0].dc_pred = j->img_comp[1].dc_pred = j->img_comp[2].dc_pred = j->img_comp[3].dc_pred =
      0;
02940    j->marker = STBI__MARKER_none;
02941    j->todo = j->restart_interval ? j->restart_interval : 0x7fffffff;
02942    j->eob_run = 0;
02943    // no more than 1<<31 MCUs if no restart_interal? that's plenty safe,
02944    // since we don't even allow 1<<30 pixels
02945 }
02946
02947 static int stbi__parse_entropy_coded_data(stbi__jpeg *z)
02948 {
02949    stbi__jpeg_reset(z);
02950    if (!z->progressive) {
02951       if (z->scan_n == 1) {
02952          int i,j;
02953          STBI_SIMD_ALIGN(short, data[64]);
02954          int n = z->order[0];
02955          // non-interleaved data, we just need to process one block at a time,
02956          // in trivial scanline order
02957          // number of blocks to do just depends on how many actual "pixels" this
02958          // component has, independent of interleaved MCU blocking and such
02959          int w = (z->img_comp[n].x+7) >> 3;
02960          int h = (z->img_comp[n].y+7) >> 3;
02961          for (j=0; j < h; ++j) {
02962             for (i=0; i < w; ++i) {
02963                int ha = z->img_comp[n].ha;
02964                if (!stbi__jpeg_decode_block(z, data, z->huff_dc+z->img_comp[n].hd, z->huff_ac+ha,
      z->fast_ac[ha], n, z->dequant[z->img_comp[n].tq])) return 0;
02965                z->idct_block_kernel(z->img_comp[n].data+z->img_comp[n].w2*j*8+i*8, z->img_comp[n].w2,
      data);
02966                // every data block is an MCU, so countdown the restart interval
02967                if (--z->todo <= 0) {
02968                   if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
02969                   // if it's NOT a restart, then just bail, so we get corrupt data
02970                   // rather than no data
02971                   if (!STBI__RESTART(z->marker)) return 1;
02972                   stbi__jpeg_reset(z);
02973                }
02974             }
02975          }
02976          return 1;
02977       } else { // interleaved
02978          int i,j,k,x,y;
02979          STBI_SIMD_ALIGN(short, data[64]);
02980          for (j=0; j < z->img_mcu_y; ++j) {
02981             for (i=0; i < z->img_mcu_x; ++i) {
02982                // scan an interleaved mcu... process scan_n components in order
02983                for (k=0; k < z->scan_n; ++k) {
02984                   int n = z->order[k];
02985                   // scan out an mcu's worth of this component; that's just determined
02986                   // by the basic H and V specified for the component
02987                   for (y=0; y < z->img_comp[n].v; ++y) {
02988                      for (x=0; x < z->img_comp[n].h; ++x) {
02989                         int x2 = (i*z->img_comp[n].h + x)*8;
02990                         int y2 = (j*z->img_comp[n].v + y)*8;
02991                         int ha = z->img_comp[n].ha;
02992                         if (!stbi__jpeg_decode_block(z, data, z->huff_dc+z->img_comp[n].hd,
      z->huff_ac+ha, z->fast_ac[ha], n, z->dequant[z->img_comp[n].tq])) return 0;
02993                         z->idct_block_kernel(z->img_comp[n].data+z->img_comp[n].w2*y2+x2,
      z->img_comp[n].w2, data);
02994                      }
02995                   }
02996                }
02997                // after all interleaved components, that's an interleaved MCU,
02998                // so now count down the restart interval
02999                if (--z->todo <= 0) {
```

```
03000                   if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
03001                   if (!STBI__RESTART(z->marker)) return 1;
03002                   stbi__jpeg_reset(z);
03003                }
03004             }
03005          }
03006          return 1;
03007       }
03008    } else {
03009       if (z->scan_n == 1) {
03010          int i,j;
03011          int n = z->order[0];
03012          // non-interleaved data, we just need to process one block at a time,
03013          // in trivial scanline order
03014          // number of blocks to do just depends on how many actual "pixels" this
03015          // component has, independent of interleaved MCU blocking and such
03016          int w = (z->img_comp[n].x+7) » 3;
03017          int h = (z->img_comp[n].y+7) » 3;
03018          for (j=0; j < h; ++j) {
03019             for (i=0; i < w; ++i) {
03020                short *data = z->img_comp[n].coeff + 64 * (i + j * z->img_comp[n].coeff_w);
03021                if (z->spec_start == 0) {
03022                   if (!stbi__jpeg_decode_block_prog_dc(z, data, &z->huff_dc[z->img_comp[n].hd], n))
03023                      return 0;
03024                } else {
03025                   int ha = z->img_comp[n].ha;
03026                   if (!stbi__jpeg_decode_block_prog_ac(z, data, &z->huff_ac[ha], z->fast_ac[ha]))
03027                      return 0;
03028                }
03029                // every data block is an MCU, so countdown the restart interval
03030                if (--z->todo <= 0) {
03031                   if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
03032                   if (!STBI__RESTART(z->marker)) return 1;
03033                   stbi__jpeg_reset(z);
03034                }
03035             }
03036          }
03037          return 1;
03038       } else { // interleaved
03039          int i,j,k,x,y;
03040          for (j=0; j < z->img_mcu_y; ++j) {
03041             for (i=0; i < z->img_mcu_x; ++i) {
03042                // scan an interleaved mcu... process scan_n components in order
03043                for (k=0; k < z->scan_n; ++k) {
03044                   int n = z->order[k];
03045                   // scan out an mcu's worth of this component; that's just determined
03046                   // by the basic H and V specified for the component
03047                   for (y=0; y < z->img_comp[n].v; ++y) {
03048                      for (x=0; x < z->img_comp[n].h; ++x) {
03049                         int x2 = (i*z->img_comp[n].h + x);
03050                         int y2 = (j*z->img_comp[n].v + y);
03051                         short *data = z->img_comp[n].coeff + 64 * (x2 + y2 * z->img_comp[n].coeff_w);
03052                         if (!stbi__jpeg_decode_block_prog_dc(z, data, &z->huff_dc[z->img_comp[n].hd],
      n))
03053                            return 0;
03054                      }
03055                   }
03056                }
03057                // after all interleaved components, that's an interleaved MCU,
03058                // so now count down the restart interval
03059                if (--z->todo <= 0) {
03060                   if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
03061                   if (!STBI__RESTART(z->marker)) return 1;
03062                   stbi__jpeg_reset(z);
03063                }
03064             }
03065          }
03066          return 1;
03067       }
03068    }
03069 }
03070
03071 static void stbi__jpeg_dequantize(short *data, stbi__uint16 *dequant)
03072 {
03073    int i;
03074    for (i=0; i < 64; ++i)
03075       data[i] *= dequant[i];
03076 }
03077
03078 static void stbi__jpeg_finish(stbi__jpeg *z)
03079 {
03080    if (z->progressive) {
03081       // dequantize and idct the data
03082       int i,j,n;
03083       for (n=0; n < z->s->img_n; ++n) {
03084          int w = (z->img_comp[n].x+7) » 3;
03085          int h = (z->img_comp[n].y+7) » 3;
```

```
03086          for (j=0; j < h; ++j) {
03087             for (i=0; i < w; ++i) {
03088                short *data = z->img_comp[n].coeff + 64 * (i + j * z->img_comp[n].coeff_w);
03089                stbi__jpeg_dequantize(data, z->dequant[z->img_comp[n].tq]);
03090                z->idct_block_kernel(z->img_comp[n].data+z->img_comp[n].w2*j*8+i*8, z->img_comp[n].w2,
      data);
03091             }
03092          }
03093       }
03094    }
03095 }
03096
03097 static int stbi__process_marker(stbi__jpeg *z, int m)
03098 {
03099    int L;
03100    switch (m) {
03101       case STBI__MARKER_none: // no marker found
03102          return stbi__err("expected marker","Corrupt JPEG");
03103
03104       case 0xDD: // DRI - specify restart interval
03105          if (stbi__get16be(z->s) != 4) return stbi__err("bad DRI len","Corrupt JPEG");
03106          z->restart_interval = stbi__get16be(z->s);
03107          return 1;
03108
03109       case 0xDB: // DQT - define quantization table
03110          L = stbi__get16be(z->s)-2;
03111          while (L > 0) {
03112             int q = stbi__get8(z->s);
03113             int p = q » 4, sixteen = (p != 0);
03114             int t = q & 15,i;
03115             if (p != 0 && p != 1) return stbi__err("bad DQT type","Corrupt JPEG");
03116             if (t > 3) return stbi__err("bad DQT table","Corrupt JPEG");
03117
03118             for (i=0; i < 64; ++i)
03119                z->dequant[t][stbi__jpeg_dezigzag[i]] = (stbi__uint16)(sixteen ? stbi__get16be(z->s) :
      stbi__get8(z->s));
03120             L -= (sixteen ? 129 : 65);
03121          }
03122          return L==0;
03123
03124       case 0xC4: // DHT - define huffman table
03125          L = stbi__get16be(z->s)-2;
03126          while (L > 0) {
03127             stbi_uc *v;
03128             int sizes[16],i,n=0;
03129             int q = stbi__get8(z->s);
03130             int tc = q » 4;
03131             int th = q & 15;
03132             if (tc > 1 || th > 3) return stbi__err("bad DHT header","Corrupt JPEG");
03133             for (i=0; i < 16; ++i) {
03134                sizes[i] = stbi__get8(z->s);
03135                n += sizes[i];
03136             }
03137             if(n > 256) return stbi__err("bad DHT header","Corrupt JPEG"); // Loop over i < n would
      write past end of values!
03138             L -= 17;
03139             if (tc == 0) {
03140                if (!stbi__build_huffman(z->huff_dc+th, sizes)) return 0;
03141                v = z->huff_dc[th].values;
03142             } else {
03143                if (!stbi__build_huffman(z->huff_ac+th, sizes)) return 0;
03144                v = z->huff_ac[th].values;
03145             }
03146             for (i=0; i < n; ++i)
03147                v[i] = stbi__get8(z->s);
03148             if (tc != 0)
03149                stbi__build_fast_ac(z->fast_ac[th], z->huff_ac + th);
03150             L -= n;
03151          }
03152          return L==0;
03153    }
03154
03155    // check for comment block or APP blocks
03156    if ((m >= 0xE0 && m <= 0xEF) || m == 0xFE) {
03157       L = stbi__get16be(z->s);
03158       if (L < 2) {
03159          if (m == 0xFE)
03160             return stbi__err("bad COM len","Corrupt JPEG");
03161          else
03162             return stbi__err("bad APP len","Corrupt JPEG");
03163       }
03164       L -= 2;
03165
03166       if (m == 0xE0 && L >= 5) { // JFIF APP0 segment
03167          static const unsigned char tag[5] = {'J','F','I','F','\0'};
03168          int ok = 1;
03169          int i;
```

```
03170          for (i=0; i < 5; ++i)
03171             if (stbi__get8(z->s) != tag[i])
03172                ok = 0;
03173          L -= 5;
03174          if (ok)
03175             z->jfif = 1;
03176       } else if (m == 0xEE && L >= 12) { // Adobe APP14 segment
03177          static const unsigned char tag[6] = {'A','d','o','b','e','\0'};
03178          int ok = 1;
03179          int i;
03180          for (i=0; i < 6; ++i)
03181             if (stbi__get8(z->s) != tag[i])
03182                ok = 0;
03183          L -= 6;
03184          if (ok) {
03185             stbi__get8(z->s); // version
03186             stbi__get16be(z->s); // flags0
03187             stbi__get16be(z->s); // flags1
03188             z->app14_color_transform = stbi__get8(z->s); // color transform
03189             L -= 6;
03190          }
03191       }
03192
03193       stbi__skip(z->s, L);
03194       return 1;
03195    }
03196
03197    return stbi__err("unknown marker","Corrupt JPEG");
03198 }
03199
03200 // after we see SOS
03201 static int stbi__process_scan_header(stbi__jpeg *z)
03202 {
03203    int i;
03204    int Ls = stbi__get16be(z->s);
03205    z->scan_n = stbi__get8(z->s);
03206    if (z->scan_n < 1 || z->scan_n > 4 || z->scan_n > (int) z->s->img_n) return stbi__err("bad SOS
       component count","Corrupt JPEG");
03207    if (Ls != 6+2*z->scan_n) return stbi__err("bad SOS len","Corrupt JPEG");
03208    for (i=0; i < z->scan_n; ++i) {
03209       int id = stbi__get8(z->s), which;
03210       int q = stbi__get8(z->s);
03211       for (which = 0; which < z->s->img_n; ++which)
03212          if (z->img_comp[which].id == id)
03213             break;
03214       if (which == z->s->img_n) return 0; // no match
03215       z->img_comp[which].hd = q >> 4;   if (z->img_comp[which].hd > 3) return stbi__err("bad DC
       huff","Corrupt JPEG");
03216       z->img_comp[which].ha = q & 15;   if (z->img_comp[which].ha > 3) return stbi__err("bad AC
       huff","Corrupt JPEG");
03217       z->order[i] = which;
03218    }
03219
03220    {
03221       int aa;
03222       z->spec_start = stbi__get8(z->s);
03223       z->spec_end   = stbi__get8(z->s); // should be 63, but might be 0
03224       aa = stbi__get8(z->s);
03225       z->succ_high = (aa >> 4);
03226       z->succ_low  = (aa & 15);
03227       if (z->progressive) {
03228          if (z->spec_start > 63 || z->spec_end > 63  || z->spec_start > z->spec_end || z->succ_high >
       13 || z->succ_low > 13)
03229             return stbi__err("bad SOS", "Corrupt JPEG");
03230       } else {
03231          if (z->spec_start != 0) return stbi__err("bad SOS","Corrupt JPEG");
03232          if (z->succ_high != 0 || z->succ_low != 0) return stbi__err("bad SOS","Corrupt JPEG");
03233          z->spec_end = 63;
03234       }
03235    }
03236
03237    return 1;
03238 }
03239
03240 static int stbi__free_jpeg_components(stbi__jpeg *z, int ncomp, int why)
03241 {
03242    int i;
03243    for (i=0; i < ncomp; ++i) {
03244       if (z->img_comp[i].raw_data) {
03245          STBI_FREE(z->img_comp[i].raw_data);
03246          z->img_comp[i].raw_data = NULL;
03247          z->img_comp[i].data = NULL;
03248       }
03249       if (z->img_comp[i].raw_coeff) {
03250          STBI_FREE(z->img_comp[i].raw_coeff);
03251          z->img_comp[i].raw_coeff = 0;
03252          z->img_comp[i].coeff = 0;
```

```
03253       }
03254       if (z->img_comp[i].linebuf) {
03255          STBI_FREE(z->img_comp[i].linebuf);
03256          z->img_comp[i].linebuf = NULL;
03257       }
03258    }
03259    return why;
03260 }
03261
03262 static int stbi__process_frame_header(stbi__jpeg *z, int scan)
03263 {
03264    stbi__context *s = z->s;
03265    int Lf,p,i,q, h_max=1,v_max=1,c;
03266    Lf = stbi__get16be(s);         if (Lf < 11) return stbi__err("bad SOF len","Corrupt JPEG"); // JPEG
03267    p  = stbi__get8(s);            if (p != 8) return stbi__err("only 8-bit","JPEG format not
      supported: 8-bit only"); // JPEG baseline
03268    s->img_y = stbi__get16be(s);   if (s->img_y == 0) return stbi__err("no header height", "JPEG format
      not supported: delayed height"); // Legal, but we don't handle it--but neither does IJG
03269    s->img_x = stbi__get16be(s);   if (s->img_x == 0) return stbi__err("0 width","Corrupt JPEG"); //
      JPEG requires
03270    if (s->img_y > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image (corrupt?)");
03271    if (s->img_x > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image (corrupt?)");
03272    c = stbi__get8(s);
03273    if (c != 3 && c != 1 && c != 4) return stbi__err("bad component count","Corrupt JPEG");
03274    s->img_n = c;
03275    for (i=0; i < c; ++i) {
03276       z->img_comp[i].data = NULL;
03277       z->img_comp[i].linebuf = NULL;
03278    }
03279
03280    if (Lf != 8+3*s->img_n) return stbi__err("bad SOF len","Corrupt JPEG");
03281
03282    z->rgb = 0;
03283    for (i=0; i < s->img_n; ++i) {
03284       static const unsigned char rgb[3] = { 'R', 'G', 'B' };
03285       z->img_comp[i].id = stbi__get8(s);
03286       if (s->img_n == 3 && z->img_comp[i].id == rgb[i])
03287          ++z->rgb;
03288       q = stbi__get8(s);
03289       z->img_comp[i].h = (q » 4);  if (!z->img_comp[i].h || z->img_comp[i].h > 4) return
      stbi__err("bad H","Corrupt JPEG");
03290       z->img_comp[i].v = q & 15;    if (!z->img_comp[i].v || z->img_comp[i].v > 4) return
      stbi__err("bad V","Corrupt JPEG");
03291       z->img_comp[i].tq = stbi__get8(s);  if (z->img_comp[i].tq > 3) return stbi__err("bad
      TQ","Corrupt JPEG");
03292    }
03293
03294    if (scan != STBI__SCAN_load) return 1;
03295
03296    if (!stbi__mad3sizes_valid(s->img_x, s->img_y, s->img_n, 0)) return stbi__err("too large", "Image
      too large to decode");
03297
03298    for (i=0; i < s->img_n; ++i) {
03299       if (z->img_comp[i].h > h_max) h_max = z->img_comp[i].h;
03300       if (z->img_comp[i].v > v_max) v_max = z->img_comp[i].v;
03301    }
03302
03303    // check that plane subsampling factors are integer ratios; our resamplers can't deal with
      fractional ratios
03304    // and I've never seen a non-corrupted JPEG file actually use them
03305    for (i=0; i < s->img_n; ++i) {
03306       if (h_max % z->img_comp[i].h != 0) return stbi__err("bad H","Corrupt JPEG");
03307       if (v_max % z->img_comp[i].v != 0) return stbi__err("bad V","Corrupt JPEG");
03308    }
03309
03310    // compute interleaved mcu info
03311    z->img_h_max = h_max;
03312    z->img_v_max = v_max;
03313    z->img_mcu_w = h_max * 8;
03314    z->img_mcu_h = v_max * 8;
03315    // these sizes can't be more than 17 bits
03316    z->img_mcu_x = (s->img_x + z->img_mcu_w-1) / z->img_mcu_w;
03317    z->img_mcu_y = (s->img_y + z->img_mcu_h-1) / z->img_mcu_h;
03318
03319    for (i=0; i < s->img_n; ++i) {
03320       // number of effective pixels (e.g. for non-interleaved MCU)
03321       z->img_comp[i].x = (s->img_x * z->img_comp[i].h + h_max-1) / h_max;
03322       z->img_comp[i].y = (s->img_y * z->img_comp[i].v + v_max-1) / v_max;
03323       // to simplify generation, we'll allocate enough memory to decode
03324       // the bogus oversized data from using interleaved MCUs and their
03325       // big blocks (e.g. a 16x16 iMCU on an image of width 33); we won't
03326       // discard the extra data until colorspace conversion
03327       //
03328       // img_mcu_x, img_mcu_y: <=17 bits; comp[i].h and .v are <=4 (checked earlier)
03329       // so these muls can't overflow with 32-bit ints (which we require)
03330       z->img_comp[i].w2 = z->img_mcu_x * z->img_comp[i].h * 8;
03331       z->img_comp[i].h2 = z->img_mcu_y * z->img_comp[i].v * 8;
```

```
03332       z->img_comp[i].coeff = 0;
03333       z->img_comp[i].raw_coeff = 0;
03334       z->img_comp[i].linebuf = NULL;
03335       z->img_comp[i].raw_data = stbi__malloc_mad2(z->img_comp[i].w2, z->img_comp[i].h2, 15);
03336       if (z->img_comp[i].raw_data == NULL)
03337          return stbi__free_jpeg_components(z, i+1, stbi__err("outofmem", "Out of memory"));
03338       // align blocks for idct using mmx/sse
03339       z->img_comp[i].data = (stbi_uc*) (((size_t) z->img_comp[i].raw_data + 15) & ~15);
03340       if (z->progressive) {
03341          // w2, h2 are multiples of 8 (see above)
03342          z->img_comp[i].coeff_w = z->img_comp[i].w2 / 8;
03343          z->img_comp[i].coeff_h = z->img_comp[i].h2 / 8;
03344          z->img_comp[i].raw_coeff = stbi__malloc_mad3(z->img_comp[i].w2, z->img_comp[i].h2,
      sizeof(short), 15);
03345          if (z->img_comp[i].raw_coeff == NULL)
03346             return stbi__free_jpeg_components(z, i+1, stbi__err("outofmem", "Out of memory"));
03347          z->img_comp[i].coeff = (short*) (((size_t) z->img_comp[i].raw_coeff + 15) & ~15);
03348       }
03349    }
03350
03351    return 1;
03352 }
03353
03354 // use comparisons since in some cases we handle more than one case (e.g. SOF)
03355 #define stbi__DNL(x)         ((x) == 0xdc)
03356 #define stbi__SOI(x)         ((x) == 0xd8)
03357 #define stbi__EOI(x)         ((x) == 0xd9)
03358 #define stbi__SOF(x)         ((x) == 0xc0 || (x) == 0xc1 || (x) == 0xc2)
03359 #define stbi__SOS(x)         ((x) == 0xda)
03360
03361 #define stbi__SOF_progressive(x)   ((x) == 0xc2)
03362
03363 static int stbi__decode_jpeg_header(stbi__jpeg *z, int scan)
03364 {
03365    int m;
03366    z->jfif = 0;
03367    z->app14_color_transform = -1; // valid values are 0,1,2
03368    z->marker = STBI__MARKER_none; // initialize cached marker to empty
03369    m = stbi__get_marker(z);
03370    if (!stbi__SOI(m)) return stbi__err("no SOI","Corrupt JPEG");
03371    if (scan == STBI__SCAN_type) return 1;
03372    m = stbi__get_marker(z);
03373    while (!stbi__SOF(m)) {
03374       if (!stbi__process_marker(z,m)) return 0;
03375       m = stbi__get_marker(z);
03376       while (m == STBI__MARKER_none) {
03377          // some files have extra padding after their blocks, so ok, we'll scan
03378          if (stbi__at_eof(z->s)) return stbi__err("no SOF", "Corrupt JPEG");
03379          m = stbi__get_marker(z);
03380       }
03381    }
03382    z->progressive = stbi__SOF_progressive(m);
03383    if (!stbi__process_frame_header(z, scan)) return 0;
03384    return 1;
03385 }
03386
03387 static int stbi__skip_jpeg_junk_at_end(stbi__jpeg *j)
03388 {
03389    // some JPEGs have junk at end, skip over it but if we find what looks
03390    // like a valid marker, resume there
03391    while (!stbi__at_eof(j->s)) {
03392       int x = stbi__get8(j->s);
03393       while (x == 255) { // might be a marker
03394          if (stbi__at_eof(j->s)) return STBI__MARKER_none;
03395          x = stbi__get8(j->s);
03396          if (x != 0x00 && x != 0xff) {
03397             // not a stuffed zero or lead-in to another marker, looks
03398             // like an actual marker, return it
03399             return x;
03400          }
03401          // stuffed zero has x=0 now which ends the loop, meaning we go
03402          // back to regular scan loop.
03403          // repeated 0xff keeps trying to read the next byte of the marker.
03404       }
03405    }
03406    return STBI__MARKER_none;
03407 }
03408
03409 // decode image to YCbCr format
03410 static int stbi__decode_jpeg_image(stbi__jpeg *j)
03411 {
03412    int m;
03413    for (m = 0; m < 4; m++) {
03414       j->img_comp[m].raw_data = NULL;
03415       j->img_comp[m].raw_coeff = NULL;
03416    }
03417    j->restart_interval = 0;
```

```
03418    if (!stbi__decode_jpeg_header(j, STBI__SCAN_load)) return 0;
03419    m = stbi__get_marker(j);
03420    while (!stbi__EOI(m)) {
03421       if (stbi__SOS(m)) {
03422          if (!stbi__process_scan_header(j)) return 0;
03423          if (!stbi__parse_entropy_coded_data(j)) return 0;
03424          if (j->marker == STBI__MARKER_none ) {
03425          j->marker = stbi__skip_jpeg_junk_at_end(j);
03426             // if we reach eof without hitting a marker, stbi__get_marker() below will fail and we'll
       eventually return 0
03427          }
03428          m = stbi__get_marker(j);
03429          if (STBI__RESTART(m))
03430             m = stbi__get_marker(j);
03431       } else if (stbi__DNL(m)) {
03432          int Ld = stbi__get16be(j->s);
03433          stbi__uint32 NL = stbi__get16be(j->s);
03434          if (Ld != 4) return stbi__err("bad DNL len", "Corrupt JPEG");
03435          if (NL != j->s->img_y) return stbi__err("bad DNL height", "Corrupt JPEG");
03436          m = stbi__get_marker(j);
03437       } else {
03438          if (!stbi__process_marker(j, m)) return 1;
03439          m = stbi__get_marker(j);
03440       }
03441    }
03442    if (j->progressive)
03443       stbi__jpeg_finish(j);
03444    return 1;
03445 }
03446
03447 // static jfif-centered resampling (across block boundaries)
03448
03449 typedef stbi_uc *(*resample_row_func)(stbi_uc *out, stbi_uc *in0, stbi_uc *in1,
03450                                       int w, int hs);
03451
03452 #define stbi__div4(x) ((stbi_uc) ((x) » 2))
03453
03454 static stbi_uc *resample_row_1(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
03455 {
03456    STBI_NOTUSED(out);
03457    STBI_NOTUSED(in_far);
03458    STBI_NOTUSED(w);
03459    STBI_NOTUSED(hs);
03460    return in_near;
03461 }
03462
03463 static stbi_uc* stbi__resample_row_v_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
03464 {
03465    // need to generate two samples vertically for every one in input
03466    int i;
03467    STBI_NOTUSED(hs);
03468    for (i=0; i < w; ++i)
03469       out[i] = stbi__div4(3*in_near[i] + in_far[i] + 2);
03470    return out;
03471 }
03472
03473 static stbi_uc*  stbi__resample_row_h_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int
       hs)
03474 {
03475    // need to generate two samples horizontally for every one in input
03476    int i;
03477    stbi_uc *input = in_near;
03478
03479    if (w == 1) {
03480       // if only one sample, can't do any interpolation
03481       out[0] = out[1] = input[0];
03482       return out;
03483    }
03484
03485    out[0] = input[0];
03486    out[1] = stbi__div4(input[0]*3 + input[1] + 2);
03487    for (i=1; i < w-1; ++i) {
03488       int n = 3*input[i]+2;
03489       out[i*2+0] = stbi__div4(n+input[i-1]);
03490       out[i*2+1] = stbi__div4(n+input[i+1]);
03491    }
03492    out[i*2+0] = stbi__div4(input[w-2]*3 + input[w-1] + 2);
03493    out[i*2+1] = input[w-1];
03494
03495    STBI_NOTUSED(in_far);
03496    STBI_NOTUSED(hs);
03497
03498    return out;
03499 }
03500
03501 #define stbi__div16(x) ((stbi_uc) ((x) » 4))
03502
```

```
03503 static stbi_uc *stbi__resample_row_hv_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int
      hs)
03504 {
03505    // need to generate 2x2 samples for every one in input
03506    int i,t0,t1;
03507    if (w == 1) {
03508       out[0] = out[1] = stbi__div4(3*in_near[0] + in_far[0] + 2);
03509       return out;
03510    }
03511
03512    t1 = 3*in_near[0] + in_far[0];
03513    out[0] = stbi__div4(t1+2);
03514    for (i=1; i < w; ++i) {
03515       t0 = t1;
03516       t1 = 3*in_near[i]+in_far[i];
03517       out[i*2-1] = stbi__div16(3*t0 + t1 + 8);
03518       out[i*2  ] = stbi__div16(3*t1 + t0 + 8);
03519    }
03520    out[w*2-1] = stbi__div4(t1+2);
03521
03522    STBI_NOTUSED(hs);
03523
03524    return out;
03525 }
03526
03527 #if defined(STBI_SSE2) || defined(STBI_NEON)
03528 static stbi_uc *stbi__resample_row_hv_2_simd(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w,
      int hs)
03529 {
03530    // need to generate 2x2 samples for every one in input
03531    int i=0,t0,t1;
03532
03533    if (w == 1) {
03534       out[0] = out[1] = stbi__div4(3*in_near[0] + in_far[0] + 2);
03535       return out;
03536    }
03537
03538    t1 = 3*in_near[0] + in_far[0];
03539    // process groups of 8 pixels for as long as we can.
03540    // note we can't handle the last pixel in a row in this loop
03541    // because we need to handle the filter boundary conditions.
03542    for (; i < ((w-1) & ~7); i += 8) {
03543 #if defined(STBI_SSE2)
03544       // load and perform the vertical filtering pass
03545       // this uses 3*x + y = 4*x + (y - x)
03546       __m128i zero  = _mm_setzero_si128();
03547       __m128i farb  = _mm_loadl_epi64((__m128i *) (in_far + i));
03548       __m128i nearb = _mm_loadl_epi64((__m128i *) (in_near + i));
03549       __m128i farw  = _mm_unpacklo_epi8(farb, zero);
03550       __m128i nearw = _mm_unpacklo_epi8(nearb, zero);
03551       __m128i diff  = _mm_sub_epi16(farw, nearw);
03552       __m128i nears = _mm_slli_epi16(nearw, 2);
03553       __m128i curr  = _mm_add_epi16(nears, diff); // current row
03554
03555       // horizontal filter works the same based on shifted vers of current
03556       // row. "prev" is current row shifted right by 1 pixel; we need to
03557       // insert the previous pixel value (from t1).
03558       // "next" is current row shifted left by 1 pixel, with first pixel
03559       // of next block of 8 pixels added in.
03560       __m128i prv0 = _mm_slli_si128(curr, 2);
03561       __m128i nxt0 = _mm_srli_si128(curr, 2);
03562       __m128i prev = _mm_insert_epi16(prv0, t1, 0);
03563       __m128i next = _mm_insert_epi16(nxt0, 3*in_near[i+8] + in_far[i+8], 7);
03564
03565       // horizontal filter, polyphase implementation since it's convenient:
03566       // even pixels = 3*cur + prev = cur*4 + (prev - cur)
03567       // odd  pixels = 3*cur + next = cur*4 + (next - cur)
03568       // note the shared term.
03569       __m128i bias  = _mm_set1_epi16(8);
03570       __m128i curs = _mm_slli_epi16(curr, 2);
03571       __m128i prvd = _mm_sub_epi16(prev, curr);
03572       __m128i nxtd = _mm_sub_epi16(next, curr);
03573       __m128i curb = _mm_add_epi16(curs, bias);
03574       __m128i even = _mm_add_epi16(prvd, curb);
03575       __m128i odd  = _mm_add_epi16(nxtd, curb);
03576
03577       // interleave even and odd pixels, then undo scaling.
03578       __m128i int0 = _mm_unpacklo_epi16(even, odd);
03579       __m128i int1 = _mm_unpackhi_epi16(even, odd);
03580       __m128i de0  = _mm_srli_epi16(int0, 4);
03581       __m128i de1  = _mm_srli_epi16(int1, 4);
03582
03583       // pack and write output
03584       __m128i outv = _mm_packus_epi16(de0, de1);
03585       _mm_storeu_si128((__m128i *) (out + i*2), outv);
03586 #elif defined(STBI_NEON)
03587       // load and perform the vertical filtering pass
```

```
03588       // this uses 3*x + y = 4*x + (y - x)
03589       uint8x8_t farb  = vld1_u8(in_far + i);
03590       uint8x8_t nearb = vld1_u8(in_near + i);
03591       int16x8_t diff  = vreinterpretq_s16_u16(vsubl_u8(farb, nearb));
03592       int16x8_t nears = vreinterpretq_s16_u16(vshll_n_u8(nearb, 2));
03593       int16x8_t curr  = vaddq_s16(nears, diff); // current row
03594
03595       // horizontal filter works the same based on shifted vers of current
03596       // row. "prev" is current row shifted right by 1 pixel; we need to
03597       // insert the previous pixel value (from t1).
03598       // "next" is current row shifted left by 1 pixel, with first pixel
03599       // of next block of 8 pixels added in.
03600       int16x8_t prv0 = vextq_s16(curr, curr, 7);
03601       int16x8_t nxt0 = vextq_s16(curr, curr, 1);
03602       int16x8_t prev = vsetq_lane_s16(t1, prv0, 0);
03603       int16x8_t next = vsetq_lane_s16(3*in_near[i+8] + in_far[i+8], nxt0, 7);
03604
03605       // horizontal filter, polyphase implementation since it's convenient:
03606       // even pixels = 3*cur + prev = cur*4 + (prev - cur)
03607       // odd  pixels = 3*cur + next = cur*4 + (next - cur)
03608       // note the shared term.
03609       int16x8_t curs = vshlq_n_s16(curr, 2);
03610       int16x8_t prvd = vsubq_s16(prev, curr);
03611       int16x8_t nxtd = vsubq_s16(next, curr);
03612       int16x8_t even = vaddq_s16(curs, prvd);
03613       int16x8_t odd  = vaddq_s16(curs, nxtd);
03614
03615       // undo scaling and round, then store with even/odd phases interleaved
03616       uint8x8x2_t o;
03617       o.val[0] = vqrshrun_n_s16(even, 4);
03618       o.val[1] = vqrshrun_n_s16(odd,  4);
03619       vst2_u8(out + i*2, o);
03620 #endif
03621
03622       // "previous" value for next iter
03623       t1 = 3*in_near[i+7] + in_far[i+7];
03624    }
03625
03626    t0 = t1;
03627    t1 = 3*in_near[i] + in_far[i];
03628    out[i*2] = stbi__div16(3*t1 + t0 + 8);
03629
03630    for (++i; i < w; ++i) {
03631       t0 = t1;
03632       t1 = 3*in_near[i]+in_far[i];
03633       out[i*2-1] = stbi__div16(3*t0 + t1 + 8);
03634       out[i*2  ] = stbi__div16(3*t1 + t0 + 8);
03635    }
03636    out[w*2-1] = stbi__div4(t1+2);
03637
03638    STBI_NOTUSED(hs);
03639
03640    return out;
03641 }
03642 #endif
03643
03644 static stbi_uc *stbi__resample_row_generic(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int
      hs)
03645 {
03646    // resample with nearest-neighbor
03647    int i,j;
03648    STBI_NOTUSED(in_far);
03649    for (i=0; i < w; ++i)
03650       for (j=0; j < hs; ++j)
03651          out[i*hs+j] = in_near[i];
03652    return out;
03653 }
03654
03655 // this is a reduced-precision calculation of YCbCr-to-RGB introduced
03656 // to make sure the code produces the same results in both SIMD and scalar
03657 #define stbi__float2fixed(x)  (((int) ((x) * 4096.0f + 0.5f)) << 8)
03658 static void stbi__YCbCr_to_RGB_row(stbi_uc *out, const stbi_uc *y, const stbi_uc *pcb, const stbi_uc
      *pcr, int count, int step)
03659 {
03660    int i;
03661    for (i=0; i < count; ++i) {
03662       int y_fixed = (y[i] << 20) + (1<<19); // rounding
03663       int r,g,b;
03664       int cr = pcr[i] - 128;
03665       int cb = pcb[i] - 128;
03666       r = y_fixed +  cr* stbi__float2fixed(1.40200f);
03667       g = y_fixed + (cr*-stbi__float2fixed(0.71414f)) + ((cb*-stbi__float2fixed(0.34414f)) &
      0xffff0000);
03668       b = y_fixed                                   +   cb* stbi__float2fixed(1.77200f);
03669       r >>= 20;
03670       g >>= 20;
03671       b >>= 20;
```

```
03672       if ((unsigned) r > 255) { if (r < 0) r = 0; else r = 255; }
03673       if ((unsigned) g > 255) { if (g < 0) g = 0; else g = 255; }
03674       if ((unsigned) b > 255) { if (b < 0) b = 0; else b = 255; }
03675       out[0] = (stbi_uc)r;
03676       out[1] = (stbi_uc)g;
03677       out[2] = (stbi_uc)b;
03678       out[3] = 255;
03679       out += step;
03680    }
03681 }
03682
03683 #if defined(STBI_SSE2) || defined(STBI_NEON)
03684 static void stbi__YCbCr_to_RGB_simd(stbi_uc *out, stbi_uc const *y, stbi_uc const *pcb, stbi_uc const
      *pcr, int count, int step)
03685 {
03686    int i = 0;
03687
03688 #ifdef STBI_SSE2
03689    // step == 3 is pretty ugly on the final interleave, and i'm not convinced
03690    // it's useful in practice (you wouldn't use it for textures, for example).
03691    // so just accelerate step == 4 case.
03692    if (step == 4) {
03693       // this is a fairly straightforward implementation and not super-optimized.
03694       __m128i signflip  = _mm_set1_epi8(-0x80);
03695       __m128i cr_const0 = _mm_set1_epi16(   (short) ( 1.40200f*4096.0f+0.5f));
03696       __m128i cr_const1 = _mm_set1_epi16( - (short) ( 0.71414f*4096.0f+0.5f));
03697       __m128i cb_const0 = _mm_set1_epi16( - (short) ( 0.34414f*4096.0f+0.5f));
03698       __m128i cb_const1 = _mm_set1_epi16(   (short) ( 1.77200f*4096.0f+0.5f));
03699       __m128i y_bias = _mm_set1_epi8((char) (unsigned char) 128);
03700       __m128i xw = _mm_set1_epi16(255); // alpha channel
03701
03702       for (; i+7 < count; i += 8) {
03703          // load
03704          __m128i y_bytes = _mm_loadl_epi64((__m128i *) (y+i));
03705          __m128i cr_bytes = _mm_loadl_epi64((__m128i *) (pcr+i));
03706          __m128i cb_bytes = _mm_loadl_epi64((__m128i *) (pcb+i));
03707          __m128i cr_biased = _mm_xor_si128(cr_bytes, signflip); // -128
03708          __m128i cb_biased = _mm_xor_si128(cb_bytes, signflip); // -128
03709
03710          // unpack to short (and left-shift cr, cb by 8)
03711          __m128i yw  = _mm_unpacklo_epi8(y_bias, y_bytes);
03712          __m128i crw = _mm_unpacklo_epi8(_mm_setzero_si128(), cr_biased);
03713          __m128i cbw = _mm_unpacklo_epi8(_mm_setzero_si128(), cb_biased);
03714
03715          // color transform
03716          __m128i yws = _mm_srli_epi16(yw, 4);
03717          __m128i cr0 = _mm_mulhi_epi16(cr_const0, crw);
03718          __m128i cb0 = _mm_mulhi_epi16(cb_const0, cbw);
03719          __m128i cb1 = _mm_mulhi_epi16(cbw, cb_const1);
03720          __m128i cr1 = _mm_mulhi_epi16(crw, cr_const1);
03721          __m128i rws = _mm_add_epi16(cr0, yws);
03722          __m128i gwt = _mm_add_epi16(cb0, yws);
03723          __m128i bws = _mm_add_epi16(yws, cb1);
03724          __m128i gws = _mm_add_epi16(gwt, cr1);
03725
03726          // descale
03727          __m128i rw = _mm_srai_epi16(rws, 4);
03728          __m128i bw = _mm_srai_epi16(bws, 4);
03729          __m128i gw = _mm_srai_epi16(gws, 4);
03730
03731          // back to byte, set up for transpose
03732          __m128i brb = _mm_packus_epi16(rw, bw);
03733          __m128i gxb = _mm_packus_epi16(gw, xw);
03734
03735          // transpose to interleave channels
03736          __m128i t0 = _mm_unpacklo_epi8(brb, gxb);
03737          __m128i t1 = _mm_unpackhi_epi8(brb, gxb);
03738          __m128i o0 = _mm_unpacklo_epi16(t0, t1);
03739          __m128i o1 = _mm_unpackhi_epi16(t0, t1);
03740
03741          // store
03742          _mm_storeu_si128((__m128i *) (out + 0), o0);
03743          _mm_storeu_si128((__m128i *) (out + 16), o1);
03744          out += 32;
03745       }
03746    }
03747 #endif
03748
03749 #ifdef STBI_NEON
03750    // in this version, step=3 support would be easy to add. but is there demand?
03751    if (step == 4) {
03752       // this is a fairly straightforward implementation and not super-optimized.
03753       uint8x8_t signflip = vdup_n_u8(0x80);
03754       int16x8_t cr_const0 = vdupq_n_s16(   (short) ( 1.40200f*4096.0f+0.5f));
03755       int16x8_t cr_const1 = vdupq_n_s16( - (short) ( 0.71414f*4096.0f+0.5f));
03756       int16x8_t cb_const0 = vdupq_n_s16( - (short) ( 0.34414f*4096.0f+0.5f));
03757       int16x8_t cb_const1 = vdupq_n_s16(   (short) ( 1.77200f*4096.0f+0.5f));
```

```
03758
03759       for (; i+7 < count; i += 8) {
03760          // load
03761          uint8x8_t y_bytes  = vld1_u8(y + i);
03762          uint8x8_t cr_bytes = vld1_u8(pcr + i);
03763          uint8x8_t cb_bytes = vld1_u8(pcb + i);
03764          int8x8_t cr_biased = vreinterpret_s8_u8(vsub_u8(cr_bytes, signflip));
03765          int8x8_t cb_biased = vreinterpret_s8_u8(vsub_u8(cb_bytes, signflip));
03766
03767          // expand to s16
03768          int16x8_t yws = vreinterpretq_s16_u16(vshll_n_u8(y_bytes, 4));
03769          int16x8_t crw = vshll_n_s8(cr_biased, 7);
03770          int16x8_t cbw = vshll_n_s8(cb_biased, 7);
03771
03772          // color transform
03773          int16x8_t cr0 = vqdmulhq_s16(crw, cr_const0);
03774          int16x8_t cb0 = vqdmulhq_s16(cbw, cb_const0);
03775          int16x8_t cr1 = vqdmulhq_s16(crw, cr_const1);
03776          int16x8_t cb1 = vqdmulhq_s16(cbw, cb_const1);
03777          int16x8_t rws = vaddq_s16(yws, cr0);
03778          int16x8_t gws = vaddq_s16(vaddq_s16(yws, cb0), cr1);
03779          int16x8_t bws = vaddq_s16(yws, cb1);
03780
03781          // undo scaling, round, convert to byte
03782          uint8x8x4_t o;
03783          o.val[0] = vqrshrun_n_s16(rws, 4);
03784          o.val[1] = vqrshrun_n_s16(gws, 4);
03785          o.val[2] = vqrshrun_n_s16(bws, 4);
03786          o.val[3] = vdup_n_u8(255);
03787
03788          // store, interleaving r/g/b/a
03789          vst4_u8(out, o);
03790          out += 8*4;
03791       }
03792    }
03793 #endif
03794
03795    for (; i < count; ++i) {
03796       int y_fixed = (y[i] « 20) + (1«19); // rounding
03797       int r,g,b;
03798       int cr = pcr[i] - 128;
03799       int cb = pcb[i] - 128;
03800       r = y_fixed + cr* stbi__float2fixed(1.40200f);
03801       g = y_fixed + cr*-stbi__float2fixed(0.71414f) + ((cb*-stbi__float2fixed(0.34414f)) &
      0xffff0000);
03802       b = y_fixed                                   +   cb* stbi__float2fixed(1.77200f);
03803       r »= 20;
03804       g »= 20;
03805       b »= 20;
03806       if ((unsigned) r > 255) { if (r < 0) r = 0; else r = 255; }
03807       if ((unsigned) g > 255) { if (g < 0) g = 0; else g = 255; }
03808       if ((unsigned) b > 255) { if (b < 0) b = 0; else b = 255; }
03809       out[0] = (stbi_uc)r;
03810       out[1] = (stbi_uc)g;
03811       out[2] = (stbi_uc)b;
03812       out[3] = 255;
03813       out += step;
03814    }
03815 }
03816 #endif
03817
03818 // set up the kernels
03819 static void stbi__setup_jpeg(stbi__jpeg *j)
03820 {
03821    j->idct_block_kernel = stbi__idct_block;
03822    j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_row;
03823    j->resample_row_hv_2_kernel = stbi__resample_row_hv_2;
03824
03825 #ifdef STBI_SSE2
03826    if (stbi__sse2_available()) {
03827       j->idct_block_kernel = stbi__idct_simd;
03828       j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_simd;
03829       j->resample_row_hv_2_kernel = stbi__resample_row_hv_2_simd;
03830    }
03831 #endif
03832
03833 #ifdef STBI_NEON
03834    j->idct_block_kernel = stbi__idct_simd;
03835    j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_simd;
03836    j->resample_row_hv_2_kernel = stbi__resample_row_hv_2_simd;
03837 #endif
03838 }
03839
03840 // clean up the temporary component buffers
03841 static void stbi__cleanup_jpeg(stbi__jpeg *j)
03842 {
03843    stbi__free_jpeg_components(j, j->s->img_n, 0);
```

```
03844 }
03845
03846 typedef struct
03847 {
03848    resample_row_func resample;
03849    stbi_uc *line0,*line1;
03850    int hs,vs;   // expansion factor in each axis
03851    int w_lores; // horizontal pixels pre-expansion
03852    int ystep;   // how far through vertical expansion we are
03853    int ypos;    // which pre-expansion row we're on
03854 } stbi__resample;
03855
03856 // fast 0..255 * 0..255 => 0..255 rounded multiplication
03857 static stbi_uc stbi__blinn_8x8(stbi_uc x, stbi_uc y)
03858 {
03859    unsigned int t = x*y + 128;
03860    return (stbi_uc) ((t + (t >>8)) >> 8);
03861 }
03862
03863 static stbi_uc *load_jpeg_image(stbi__jpeg *z, int *out_x, int *out_y, int *comp, int req_comp)
03864 {
03865    int n, decode_n, is_rgb;
03866    z->s->img_n = 0; // make stbi__cleanup_jpeg safe
03867
03868    // validate req_comp
03869    if (req_comp < 0 || req_comp > 4) return stbi__errpuc("bad req_comp", "Internal error");
03870
03871    // load a jpeg image from whichever source, but leave in YCbCr format
03872    if (!stbi__decode_jpeg_image(z)) { stbi__cleanup_jpeg(z); return NULL; }
03873
03874    // determine actual number of components to generate
03875    n = req_comp ? req_comp : z->s->img_n >= 3 ? 3 : 1;
03876
03877    is_rgb = z->s->img_n == 3 && (z->rgb == 3 || (z->app14_color_transform == 0 && !z->jfif));
03878
03879    if (z->s->img_n == 3 && n < 3 && !is_rgb)
03880       decode_n = 1;
03881    else
03882       decode_n = z->s->img_n;
03883
03884    // nothing to do if no components requested; check this now to avoid
03885    // accessing uninitialized coutput[0] later
03886    if (decode_n <= 0) { stbi__cleanup_jpeg(z); return NULL; }
03887
03888    // resample and color-convert
03889    {
03890       int k;
03891       unsigned int i,j;
03892       stbi_uc *output;
03893       stbi_uc *coutput[4] = { NULL, NULL, NULL, NULL };
03894
03895       stbi__resample res_comp[4];
03896
03897       for (k=0; k < decode_n; ++k) {
03898          stbi__resample *r = &res_comp[k];
03899
03900          // allocate line buffer big enough for upsampling off the edges
03901          // with upsample factor of 4
03902          z->img_comp[k].linebuf = (stbi_uc *) stbi__malloc(z->s->img_x + 3);
03903          if (!z->img_comp[k].linebuf) { stbi__cleanup_jpeg(z); return stbi__errpuc("outofmem", "Out of
      memory"); }
03904
03905          r->hs      = z->img_h_max / z->img_comp[k].h;
03906          r->vs      = z->img_v_max / z->img_comp[k].v;
03907          r->ystep   = r->vs >> 1;
03908          r->w_lores = (z->s->img_x + r->hs-1) / r->hs;
03909          r->ypos    = 0;
03910          r->line0   = r->line1 = z->img_comp[k].data;
03911
03912          if      (r->hs == 1 && r->vs == 1) r->resample = resample_row_1;
03913          else if (r->hs == 1 && r->vs == 2) r->resample = stbi__resample_row_v_2;
03914          else if (r->hs == 2 && r->vs == 1) r->resample = stbi__resample_row_h_2;
03915          else if (r->hs == 2 && r->vs == 2) r->resample = z->resample_row_hv_2_kernel;
03916          else                               r->resample = stbi__resample_row_generic;
03917       }
03918
03919       // can't error after this so, this is safe
03920       output = (stbi_uc *) stbi__malloc_mad3(n, z->s->img_x, z->s->img_y, 1);
03921       if (!output) { stbi__cleanup_jpeg(z); return stbi__errpuc("outofmem", "Out of memory"); }
03922
03923       // now go ahead and resample
03924       for (j=0; j < z->s->img_y; ++j) {
03925          stbi_uc *out = output + n * z->s->img_x * j;
03926          for (k=0; k < decode_n; ++k) {
03927             stbi__resample *r = &res_comp[k];
03928             int y_bot = r->ystep >= (r->vs >> 1);
03929             coutput[k] = r->resample(z->img_comp[k].linebuf,
```

```
03930                                   y_bot ? r->line1 : r->line0,
03931                                   y_bot ? r->line0 : r->line1,
03932                                   r->w_lores, r->hs);
03933            if (++r->ystep >= r->vs) {
03934               r->ystep = 0;
03935               r->line0 = r->line1;
03936               if (++r->ypos < z->img_comp[k].y)
03937                  r->line1 += z->img_comp[k].w2;
03938            }
03939         }
03940         if (n >= 3) {
03941            stbi_uc *y = coutput[0];
03942            if (z->s->img_n == 3) {
03943               if (is_rgb) {
03944                  for (i=0; i < z->s->img_x; ++i) {
03945                     out[0] = y[i];
03946                     out[1] = coutput[1][i];
03947                     out[2] = coutput[2][i];
03948                     out[3] = 255;
03949                     out += n;
03950                  }
03951               } else {
03952                  z->YCbCr_to_RGB_kernel(out, y, coutput[1], coutput[2], z->s->img_x, n);
03953               }
03954            } else if (z->s->img_n == 4) {
03955               if (z->app14_color_transform == 0) { // CMYK
03956                  for (i=0; i < z->s->img_x; ++i) {
03957                     stbi_uc m = coutput[3][i];
03958                     out[0] = stbi__blinn_8x8(coutput[0][i], m);
03959                     out[1] = stbi__blinn_8x8(coutput[1][i], m);
03960                     out[2] = stbi__blinn_8x8(coutput[2][i], m);
03961                     out[3] = 255;
03962                     out += n;
03963                  }
03964               } else if (z->app14_color_transform == 2) { // YCCK
03965                  z->YCbCr_to_RGB_kernel(out, y, coutput[1], coutput[2], z->s->img_x, n);
03966                  for (i=0; i < z->s->img_x; ++i) {
03967                     stbi_uc m = coutput[3][i];
03968                     out[0] = stbi__blinn_8x8(255 - out[0], m);
03969                     out[1] = stbi__blinn_8x8(255 - out[1], m);
03970                     out[2] = stbi__blinn_8x8(255 - out[2], m);
03971                     out += n;
03972                  }
03973               } else { // YCbCr + alpha?  Ignore the fourth channel for now
03974                  z->YCbCr_to_RGB_kernel(out, y, coutput[1], coutput[2], z->s->img_x, n);
03975               }
03976            } else
03977               for (i=0; i < z->s->img_x; ++i) {
03978                  out[0] = out[1] = out[2] = y[i];
03979                  out[3] = 255; // not used if n==3
03980                  out += n;
03981               }
03982         } else {
03983            if (is_rgb) {
03984               if (n == 1)
03985                  for (i=0; i < z->s->img_x; ++i)
03986                     *out++ = stbi__compute_y(coutput[0][i], coutput[1][i], coutput[2][i]);
03987               else {
03988                  for (i=0; i < z->s->img_x; ++i, out += 2) {
03989                     out[0] = stbi__compute_y(coutput[0][i], coutput[1][i], coutput[2][i]);
03990                     out[1] = 255;
03991                  }
03992               }
03993            } else if (z->s->img_n == 4 && z->app14_color_transform == 0) {
03994               for (i=0; i < z->s->img_x; ++i) {
03995                  stbi_uc m = coutput[3][i];
03996                  stbi_uc r = stbi__blinn_8x8(coutput[0][i], m);
03997                  stbi_uc g = stbi__blinn_8x8(coutput[1][i], m);
03998                  stbi_uc b = stbi__blinn_8x8(coutput[2][i], m);
03999                  out[0] = stbi__compute_y(r, g, b);
04000                  out[1] = 255;
04001                  out += n;
04002               }
04003            } else if (z->s->img_n == 4 && z->app14_color_transform == 2) {
04004               for (i=0; i < z->s->img_x; ++i) {
04005                  out[0] = stbi__blinn_8x8(255 - coutput[0][i], coutput[3][i]);
04006                  out[1] = 255;
04007                  out += n;
04008               }
04009            } else {
04010               stbi_uc *y = coutput[0];
04011               if (n == 1)
04012                  for (i=0; i < z->s->img_x; ++i) out[i] = y[i];
04013               else
04014                  for (i=0; i < z->s->img_x; ++i) { *out++ = y[i]; *out++ = 255; }
04015            }
04016         }
```

```
04017       }
04018       stbi__cleanup_jpeg(z);
04019       *out_x = z->s->img_x;
04020       *out_y = z->s->img_y;
04021       if (comp) *comp = z->s->img_n >= 3 ? 3 : 1; // report original components, not output
04022       return output;
04023    }
04024 }
04025
04026 static void *stbi__jpeg_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri)
04027 {
04028    unsigned char* result;
04029    stbi__jpeg* j = (stbi__jpeg*) stbi__malloc(sizeof(stbi__jpeg));
04030    if (!j) return stbi__errpuc("outofmem", "Out of memory");
04031    memset(j, 0, sizeof(stbi__jpeg));
04032    STBI_NOTUSED(ri);
04033    j->s = s;
04034    stbi__setup_jpeg(j);
04035    result = load_jpeg_image(j, x,y,comp,req_comp);
04036    STBI_FREE(j);
04037    return result;
04038 }
04039
04040 static int stbi__jpeg_test(stbi__context *s)
04041 {
04042    int r;
04043    stbi__jpeg* j = (stbi__jpeg*)stbi__malloc(sizeof(stbi__jpeg));
04044    if (!j) return stbi__err("outofmem", "Out of memory");
04045    memset(j, 0, sizeof(stbi__jpeg));
04046    j->s = s;
04047    stbi__setup_jpeg(j);
04048    r = stbi__decode_jpeg_header(j, STBI__SCAN_type);
04049    stbi__rewind(s);
04050    STBI_FREE(j);
04051    return r;
04052 }
04053
04054 static int stbi__jpeg_info_raw(stbi__jpeg *j, int *x, int *y, int *comp)
04055 {
04056    if (!stbi__decode_jpeg_header(j, STBI__SCAN_header)) {
04057       stbi__rewind( j->s );
04058       return 0;
04059    }
04060    if (x) *x = j->s->img_x;
04061    if (y) *y = j->s->img_y;
04062    if (comp) *comp = j->s->img_n >= 3 ? 3 : 1;
04063    return 1;
04064 }
04065
04066 static int stbi__jpeg_info(stbi__context *s, int *x, int *y, int *comp)
04067 {
04068    int result;
04069    stbi__jpeg* j = (stbi__jpeg*) (stbi__malloc(sizeof(stbi__jpeg)));
04070    if (!j) return stbi__err("outofmem", "Out of memory");
04071    memset(j, 0, sizeof(stbi__jpeg));
04072    j->s = s;
04073    result = stbi__jpeg_info_raw(j, x, y, comp);
04074    STBI_FREE(j);
04075    return result;
04076 }
04077 #endif
04078
04079 // public domain zlib decode    v0.2  Sean Barrett 2006-11-18
04080 //    simple implementation
04081 //      - all input must be provided in an upfront buffer
04082 //      - all output is written to a single output buffer (can malloc/realloc)
04083 //    performance
04084 //      - fast huffman
04085
04086 #ifndef STBI_NO_ZLIB
04087
04088 // fast-way is faster to check than jpeg huffman, but slow way is slower
04089 #define STBI__ZFAST_BITS  9 // accelerate all cases in default tables
04090 #define STBI__ZFAST_MASK  ((1 « STBI__ZFAST_BITS) - 1)
04091 #define STBI__ZNSYMS 288 // number of symbols in literal/length alphabet
04092
04093 // zlib-style huffman encoding
04094 // (jpegs packs from left, zlib from right, so can't share code)
04095 typedef struct
04096 {
04097    stbi__uint16 fast[1 « STBI__ZFAST_BITS];
04098    stbi__uint16 firstcode[16];
04099    int maxcode[17];
04100    stbi__uint16 firstsymbol[16];
04101    stbi_uc  size[STBI__ZNSYMS];
04102    stbi__uint16 value[STBI__ZNSYMS];
```

```
04103 } stbi__zhuffman;
04104
04105 stbi_inline static int stbi__bitreverse16(int n)
04106 {
04107   n = ((n & 0xAAAA) >>  1) | ((n & 0x5555) << 1);
04108   n = ((n & 0xCCCC) >>  2) | ((n & 0x3333) << 2);
04109   n = ((n & 0xF0F0) >>  4) | ((n & 0x0F0F) << 4);
04110   n = ((n & 0xFF00) >>  8) | ((n & 0x00FF) << 8);
04111   return n;
04112 }
04113
04114 stbi_inline static int stbi__bit_reverse(int v, int bits)
04115 {
04116    STBI_ASSERT(bits <= 16);
04117    // to bit reverse n bits, reverse 16 and shift
04118    // e.g. 11 bits, bit reverse and shift away 5
04119    return stbi__bitreverse16(v) >> (16-bits);
04120 }
04121
04122 static int stbi__zbuild_huffman(stbi__zhuffman *z, const stbi_uc *sizelist, int num)
04123 {
04124    int i,k=0;
04125    int code, next_code[16], sizes[17];
04126
04127    // DEFLATE spec for generating codes
04128    memset(sizes, 0, sizeof(sizes));
04129    memset(z->fast, 0, sizeof(z->fast));
04130    for (i=0; i < num; ++i)
04131       ++sizes[sizelist[i]];
04132    sizes[0] = 0;
04133    for (i=1; i < 16; ++i)
04134       if (sizes[i] > (1 << i))
04135          return stbi__err("bad sizes", "Corrupt PNG");
04136    code = 0;
04137    for (i=1; i < 16; ++i) {
04138       next_code[i] = code;
04139       z->firstcode[i] = (stbi__uint16) code;
04140       z->firstsymbol[i] = (stbi__uint16) k;
04141       code = (code + sizes[i]);
04142       if (sizes[i])
04143          if (code-1 >= (1 << i)) return stbi__err("bad codelengths","Corrupt PNG");
04144       z->maxcode[i] = code << (16-i); // preshift for inner loop
04145       code <<= 1;
04146       k += sizes[i];
04147    }
04148    z->maxcode[16] = 0x10000; // sentinel
04149    for (i=0; i < num; ++i) {
04150       int s = sizelist[i];
04151       if (s) {
04152          int c = next_code[s] - z->firstcode[s] + z->firstsymbol[s];
04153          stbi__uint16 fastv = (stbi__uint16) ((s << 9) | i);
04154          z->size [c] = (stbi_uc     ) s;
04155          z->value[c] = (stbi__uint16) i;
04156          if (s <= STBI__ZFAST_BITS) {
04157             int j = stbi__bit_reverse(next_code[s],s);
04158             while (j < (1 << STBI__ZFAST_BITS)) {
04159                z->fast[j] = fastv;
04160                j += (1 << s);
04161             }
04162          }
04163          ++next_code[s];
04164       }
04165    }
04166    return 1;
04167 }
04168
04169 // zlib-from-memory implementation for PNG reading
04170 //    because PNG allows splitting the zlib stream arbitrarily,
04171 //    and it's annoying structurally to have PNG call ZLIB call PNG,
04172 //    we require PNG read all the IDATs and combine them into a single
04173 //    memory buffer
04174
04175 typedef struct
04176 {
04177    stbi_uc *zbuffer, *zbuffer_end;
04178    int num_bits;
04179    stbi__uint32 code_buffer;
04180
04181    char *zout;
04182    char *zout_start;
04183    char *zout_end;
04184    int   z_expandable;
04185
04186    stbi__zhuffman z_length, z_distance;
04187 } stbi__zbuf;
04188
04189 stbi_inline static int stbi__zeof(stbi__zbuf *z)
```

```
04190 {
04191    return (z->zbuffer >= z->zbuffer_end);
04192 }
04193
04194 stbi_inline static stbi_uc stbi__zget8(stbi__zbuf *z)
04195 {
04196    return stbi__zeof(z) ? 0 : *z->zbuffer++;
04197 }
04198
04199 static void stbi__fill_bits(stbi__zbuf *z)
04200 {
04201    do {
04202       if (z->code_buffer >= (1U << z->num_bits)) {
04203         z->zbuffer = z->zbuffer_end;  /* treat this as EOF so we fail. */
04204         return;
04205       }
04206       z->code_buffer |= (unsigned int) stbi__zget8(z) << z->num_bits;
04207       z->num_bits += 8;
04208    } while (z->num_bits <= 24);
04209 }
04210
04211 stbi_inline static unsigned int stbi__zreceive(stbi__zbuf *z, int n)
04212 {
04213    unsigned int k;
04214    if (z->num_bits < n) stbi__fill_bits(z);
04215    k = z->code_buffer & ((1 << n) - 1);
04216    z->code_buffer >>= n;
04217    z->num_bits -= n;
04218    return k;
04219 }
04220
04221 static int stbi__zhuffman_decode_slowpath(stbi__zbuf *a, stbi__zhuffman *z)
04222 {
04223    int b,s,k;
04224    // not resolved by fast table, so compute it the slow way
04225    // use jpeg approach, which requires MSbits at top
04226    k = stbi__bit_reverse(a->code_buffer, 16);
04227    for (s=STBI__ZFAST_BITS+1; ; ++s)
04228       if (k < z->maxcode[s])
04229          break;
04230    if (s >= 16) return -1; // invalid code!
04231    // code size is s, so:
04232    b = (k >> (16-s)) - z->firstcode[s] + z->firstsymbol[s];
04233    if (b >= STBI__ZNSYMS) return -1; // some data was corrupt somewhere!
04234    if (z->size[b] != s) return -1;  // was originally an assert, but report failure instead.
04235    a->code_buffer >>= s;
04236    a->num_bits -= s;
04237    return z->value[b];
04238 }
04239
04240 stbi_inline static int stbi__zhuffman_decode(stbi__zbuf *a, stbi__zhuffman *z)
04241 {
04242    int b,s;
04243    if (a->num_bits < 16) {
04244       if (stbi__zeof(a)) {
04245          return -1;   /* report error for unexpected end of data. */
04246       }
04247       stbi__fill_bits(a);
04248    }
04249    b = z->fast[a->code_buffer & STBI__ZFAST_MASK];
04250    if (b) {
04251       s = b >> 9;
04252       a->code_buffer >>= s;
04253       a->num_bits -= s;
04254       return b & 511;
04255    }
04256    return stbi__zhuffman_decode_slowpath(a, z);
04257 }
04258
04259 static int stbi__zexpand(stbi__zbuf *z, char *zout, int n)  // need to make room for n bytes
04260 {
04261    char *q;
04262    unsigned int cur, limit, old_limit;
04263    z->zout = zout;
04264    if (!z->z_expandable) return stbi__err("output buffer limit","Corrupt PNG");
04265    cur   = (unsigned int) (z->zout - z->zout_start);
04266    limit = old_limit = (unsigned) (z->zout_end - z->zout_start);
04267    if (UINT_MAX - cur < (unsigned) n) return stbi__err("outofmem", "Out of memory");
04268    while (cur + n > limit) {
04269       if(limit > UINT_MAX / 2) return stbi__err("outofmem", "Out of memory");
04270       limit *= 2;
04271    }
04272    q = (char *) STBI_REALLOC_SIZED(z->zout_start, old_limit, limit);
04273    STBI_NOTUSED(old_limit);
04274    if (q == NULL) return stbi__err("outofmem", "Out of memory");
04275    z->zout_start = q;
04276    z->zout       = q + cur;
```

```
04277    z->zout_end   = q + limit;
04278    return 1;
04279 }
04280
04281 static const int stbi__zlength_base[31] = {
04282    3,4,5,6,7,8,9,10,11,13,
04283    15,17,19,23,27,31,35,43,51,59,
04284    67,83,99,115,131,163,195,227,258,0,0 };
04285
04286 static const int stbi__zlength_extra[31]=
04287 { 0,0,0,0,0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,0,0,0 };
04288
04289 static const int stbi__zdist_base[32] = { 1,2,3,4,5,7,9,13,17,25,33,49,65,97,129,193,
04290 257,385,513,769,1025,1537,2049,3073,4097,6145,8193,12289,16385,24577,0,0};
04291
04292 static const int stbi__zdist_extra[32] =
04293 { 0,0,0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,12,13,13};
04294
04295 static int stbi__parse_huffman_block(stbi__zbuf *a)
04296 {
04297    char *zout = a->zout;
04298    for(;;) {
04299       int z = stbi__zhuffman_decode(a, &a->z_length);
04300       if (z < 256) {
04301          if (z < 0) return stbi__err("bad huffman code","Corrupt PNG"); // error in huffman codes
04302          if (zout >= a->zout_end) {
04303             if (!stbi__zexpand(a, zout, 1)) return 0;
04304             zout = a->zout;
04305          }
04306          *zout++ = (char) z;
04307       } else {
04308          stbi_uc *p;
04309          int len,dist;
04310          if (z == 256) {
04311             a->zout = zout;
04312             return 1;
04313          }
04314          if (z >= 286) return stbi__err("bad huffman code","Corrupt PNG"); // per DEFLATE, length
      codes 286 and 287 must not appear in compressed data
04315          z -= 257;
04316          len = stbi__zlength_base[z];
04317          if (stbi__zlength_extra[z]) len += stbi__zreceive(a, stbi__zlength_extra[z]);
04318          z = stbi__zhuffman_decode(a, &a->z_distance);
04319          if (z < 0 || z >= 30) return stbi__err("bad huffman code","Corrupt PNG"); // per DEFLATE,
      distance codes 30 and 31 must not appear in compressed data
04320          dist = stbi__zdist_base[z];
04321          if (stbi__zdist_extra[z]) dist += stbi__zreceive(a, stbi__zdist_extra[z]);
04322          if (zout - a->zout_start < dist) return stbi__err("bad dist","Corrupt PNG");
04323          if (zout + len > a->zout_end) {
04324             if (!stbi__zexpand(a, zout, len)) return 0;
04325             zout = a->zout;
04326          }
04327          p = (stbi_uc *) (zout - dist);
04328          if (dist == 1) { // run of one byte; common in images.
04329             stbi_uc v = *p;
04330             if (len) { do *zout++ = v; while (--len); }
04331          } else {
04332             if (len) { do *zout++ = *p++; while (--len); }
04333          }
04334       }
04335    }
04336 }
04337
04338 static int stbi__compute_huffman_codes(stbi__zbuf *a)
04339 {
04340    static const stbi_uc length_dezigzag[19] = { 16,17,18,0,8,7,9,6,10,5,11,4,12,3,13,2,14,1,15 };
04341    stbi__zhuffman z_codelength;
04342    stbi_uc lencodes[286+32+137];//padding for maximum single op
04343    stbi_uc codelength_sizes[19];
04344    int i,n;
04345
04346    int hlit  = stbi__zreceive(a,5) + 257;
04347    int hdist = stbi__zreceive(a,5) + 1;
04348    int hclen = stbi__zreceive(a,4) + 4;
04349    int ntot  = hlit + hdist;
04350
04351    memset(codelength_sizes, 0, sizeof(codelength_sizes));
04352    for (i=0; i < hclen; ++i) {
04353       int s = stbi__zreceive(a,3);
04354       codelength_sizes[length_dezigzag[i]] = (stbi_uc) s;
04355    }
04356    if (!stbi__zbuild_huffman(&z_codelength, codelength_sizes, 19)) return 0;
04357
04358    n = 0;
04359    while (n < ntot) {
04360       int c = stbi__zhuffman_decode(a, &z_codelength);
04361       if (c < 0 || c >= 19) return stbi__err("bad codelengths", "Corrupt PNG");
```

```
04362       if (c < 16)
04363          lencodes[n++] = (stbi_uc) c;
04364       else {
04365          stbi_uc fill = 0;
04366          if (c == 16) {
04367             c = stbi__zreceive(a,2)+3;
04368             if (n == 0) return stbi__err("bad codelengths", "Corrupt PNG");
04369             fill = lencodes[n-1];
04370          } else if (c == 17) {
04371             c = stbi__zreceive(a,3)+3;
04372          } else if (c == 18) {
04373             c = stbi__zreceive(a,7)+11;
04374          } else {
04375             return stbi__err("bad codelengths", "Corrupt PNG");
04376          }
04377          if (ntot - n < c) return stbi__err("bad codelengths", "Corrupt PNG");
04378          memset(lencodes+n, fill, c);
04379          n += c;
04380       }
04381    }
04382    if (n != ntot) return stbi__err("bad codelengths","Corrupt PNG");
04383    if (!stbi__zbuild_huffman(&a->z_length, lencodes, hlit)) return 0;
04384    if (!stbi__zbuild_huffman(&a->z_distance, lencodes+hlit, hdist)) return 0;
04385    return 1;
04386 }
04387
04388 static int stbi__parse_uncompressed_block(stbi__zbuf *a)
04389 {
04390    stbi_uc header[4];
04391    int len,nlen,k;
04392    if (a->num_bits & 7)
04393       stbi__zreceive(a, a->num_bits & 7); // discard
04394    // drain the bit-packed data into header
04395    k = 0;
04396    while (a->num_bits > 0) {
04397       header[k++] = (stbi_uc) (a->code_buffer & 255); // suppress MSVC run-time check
04398       a->code_buffer >>= 8;
04399       a->num_bits -= 8;
04400    }
04401    if (a->num_bits < 0) return stbi__err("zlib corrupt","Corrupt PNG");
04402    // now fill header the normal way
04403    while (k < 4)
04404       header[k++] = stbi__zget8(a);
04405    len  = header[1] * 256 + header[0];
04406    nlen = header[3] * 256 + header[2];
04407    if (nlen != (len ^ 0xffff)) return stbi__err("zlib corrupt","Corrupt PNG");
04408    if (a->zbuffer + len > a->zbuffer_end) return stbi__err("read past buffer","Corrupt PNG");
04409    if (a->zout + len > a->zout_end)
04410       if (!stbi__zexpand(a, a->zout, len)) return 0;
04411    memcpy(a->zout, a->zbuffer, len);
04412    a->zbuffer += len;
04413    a->zout += len;
04414    return 1;
04415 }
04416
04417 static int stbi__parse_zlib_header(stbi__zbuf *a)
04418 {
04419    int cmf   = stbi__zget8(a);
04420    int cm    = cmf & 15;
04421    /* int cinfo = cmf >> 4; */
04422    int flg   = stbi__zget8(a);
04423    if (stbi__zeof(a)) return stbi__err("bad zlib header","Corrupt PNG"); // zlib spec
04424    if ((cmf*256+flg) % 31 != 0) return stbi__err("bad zlib header","Corrupt PNG"); // zlib spec
04425    if (flg & 32) return stbi__err("no preset dict","Corrupt PNG"); // preset dictionary not allowed in
       png
04426    if (cm != 8) return stbi__err("bad compression","Corrupt PNG"); // DEFLATE required for png
04427    // window = 1 << (8 + cinfo)... but who cares, we fully buffer output
04428    return 1;
04429 }
04430
04431 static const stbi_uc stbi__zdefault_length[STBI__ZNSYMS] =
04432 {
04433    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
04434    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
04435    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
04436    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
04437    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
04438    9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9, 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
04439    9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9, 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
04440    9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9, 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
04441    7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7, 7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8
04442 };
04443 static const stbi_uc stbi__zdefault_distance[32] =
04444 {
04445    5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5
04446 };
04447 /*
```

```
04448 Init algorithm:
04449 {
04450    int i;   // use <= to match clearly with spec
04451    for (i=0; i <= 143; ++i)     stbi__zdefault_length[i]   = 8;
04452    for (   ; i <= 255; ++i)     stbi__zdefault_length[i]   = 9;
04453    for (   ; i <= 279; ++i)     stbi__zdefault_length[i]   = 7;
04454    for (   ; i <= 287; ++i)     stbi__zdefault_length[i]   = 8;
04455
04456    for (i=0; i <=  31; ++i)     stbi__zdefault_distance[i] = 5;
04457 }
04458 */
04459
04460 static int stbi__parse_zlib(stbi__zbuf *a, int parse_header)
04461 {
04462    int final, type;
04463    if (parse_header)
04464       if (!stbi__parse_zlib_header(a)) return 0;
04465    a->num_bits = 0;
04466    a->code_buffer = 0;
04467    do {
04468       final = stbi__zreceive(a,1);
04469       type = stbi__zreceive(a,2);
04470       if (type == 0) {
04471          if (!stbi__parse_uncompressed_block(a)) return 0;
04472       } else if (type == 3) {
04473          return 0;
04474       } else {
04475          if (type == 1) {
04476             // use fixed code lengths
04477             if (!stbi__zbuild_huffman(&a->z_length  , stbi__zdefault_length  , STBI__ZNSYMS)) return
      0;
04478             if (!stbi__zbuild_huffman(&a->z_distance, stbi__zdefault_distance,  32)) return 0;
04479          } else {
04480             if (!stbi__compute_huffman_codes(a)) return 0;
04481          }
04482          if (!stbi__parse_huffman_block(a)) return 0;
04483       }
04484    } while (!final);
04485    return 1;
04486 }
04487
04488 static int stbi__do_zlib(stbi__zbuf *a, char *obuf, int olen, int exp, int parse_header)
04489 {
04490    a->zout_start = obuf;
04491    a->zout       = obuf;
04492    a->zout_end   = obuf + olen;
04493    a->z_expandable = exp;
04494
04495    return stbi__parse_zlib(a, parse_header);
04496 }
04497
04498 STBIDEF char *stbi_zlib_decode_malloc_guesssize(const char *buffer, int len, int initial_size, int
      *outlen)
04499 {
04500    stbi__zbuf a;
04501    char *p = (char *) stbi__malloc(initial_size);
04502    if (p == NULL) return NULL;
04503    a.zbuffer = (stbi_uc *) buffer;
04504    a.zbuffer_end = (stbi_uc *) buffer + len;
04505    if (stbi__do_zlib(&a, p, initial_size, 1, 1)) {
04506       if (outlen) *outlen = (int) (a.zout - a.zout_start);
04507       return a.zout_start;
04508    } else {
04509       STBI_FREE(a.zout_start);
04510       return NULL;
04511    }
04512 }
04513
04514 STBIDEF char *stbi_zlib_decode_malloc(char const *buffer, int len, int *outlen)
04515 {
04516    return stbi_zlib_decode_malloc_guesssize(buffer, len, 16384, outlen);
04517 }
04518
04519 STBIDEF char *stbi_zlib_decode_malloc_guesssize_headerflag(const char *buffer, int len, int
      initial_size, int *outlen, int parse_header)
04520 {
04521    stbi__zbuf a;
04522    char *p = (char *) stbi__malloc(initial_size);
04523    if (p == NULL) return NULL;
04524    a.zbuffer = (stbi_uc *) buffer;
04525    a.zbuffer_end = (stbi_uc *) buffer + len;
04526    if (stbi__do_zlib(&a, p, initial_size, 1, parse_header)) {
04527       if (outlen) *outlen = (int) (a.zout - a.zout_start);
04528       return a.zout_start;
04529    } else {
04530       STBI_FREE(a.zout_start);
04531       return NULL;
```

```
04532    }
04533 }
04534
04535 STBIDEF int stbi_zlib_decode_buffer(char *obuffer, int olen, char const *ibuffer, int ilen)
04536 {
04537    stbi__zbuf a;
04538    a.zbuffer = (stbi_uc *) ibuffer;
04539    a.zbuffer_end = (stbi_uc *) ibuffer + ilen;
04540    if (stbi__do_zlib(&a, obuffer, olen, 0, 1))
04541       return (int) (a.zout - a.zout_start);
04542    else
04543       return -1;
04544 }
04545
04546 STBIDEF char *stbi_zlib_decode_noheader_malloc(char const *buffer, int len, int *outlen)
04547 {
04548    stbi__zbuf a;
04549    char *p = (char *) stbi__malloc(16384);
04550    if (p == NULL) return NULL;
04551    a.zbuffer = (stbi_uc *) buffer;
04552    a.zbuffer_end = (stbi_uc *) buffer+len;
04553    if (stbi__do_zlib(&a, p, 16384, 1, 0)) {
04554       if (outlen) *outlen = (int) (a.zout - a.zout_start);
04555       return a.zout_start;
04556    } else {
04557       STBI_FREE(a.zout_start);
04558       return NULL;
04559    }
04560 }
04561
04562 STBIDEF int stbi_zlib_decode_noheader_buffer(char *obuffer, int olen, const char *ibuffer, int ilen)
04563 {
04564    stbi__zbuf a;
04565    a.zbuffer = (stbi_uc *) ibuffer;
04566    a.zbuffer_end = (stbi_uc *) ibuffer + ilen;
04567    if (stbi__do_zlib(&a, obuffer, olen, 0, 0))
04568       return (int) (a.zout - a.zout_start);
04569    else
04570       return -1;
04571 }
04572 #endif
04573
04574 // public domain "baseline" PNG decoder   v0.10  Sean Barrett 2006-11-18
04575 //    simple implementation
04576 //      - only 8-bit samples
04577 //      - no CRC checking
04578 //      - allocates lots of intermediate memory
04579 //        - avoids problem of streaming data between subsystems
04580 //        - avoids explicit window management
04581 //    performance
04582 //      - uses stb_zlib, a PD zlib implementation with fast huffman decoding
04583
04584 #ifndef STBI_NO_PNG
04585 typedef struct
04586 {
04587    stbi__uint32 length;
04588    stbi__uint32 type;
04589 } stbi__pngchunk;
04590
04591 static stbi__pngchunk stbi__get_chunk_header(stbi__context *s)
04592 {
04593    stbi__pngchunk c;
04594    c.length = stbi__get32be(s);
04595    c.type   = stbi__get32be(s);
04596    return c;
04597 }
04598
04599 static int stbi__check_png_header(stbi__context *s)
04600 {
04601    static const stbi_uc png_sig[8] = { 137,80,78,71,13,10,26,10 };
04602    int i;
04603    for (i=0; i < 8; ++i)
04604       if (stbi__get8(s) != png_sig[i]) return stbi__err("bad png sig","Not a PNG");
04605    return 1;
04606 }
04607
04608 typedef struct
04609 {
04610    stbi__context *s;
04611    stbi_uc *idata, *expanded, *out;
04612    int depth;
04613 } stbi__png;
04614
04615
04616 enum {
04617    STBI__F_none=0,
04618    STBI__F_sub=1,
```

```
04619    STBI__F_up=2,
04620    STBI__F_avg=3,
04621    STBI__F_paeth=4,
04622    // synthetic filters used for first scanline to avoid needing a dummy row of 0s
04623    STBI__F_avg_first,
04624    STBI__F_paeth_first
04625 };
04626
04627 static stbi_uc first_row_filter[5] =
04628 {
04629    STBI__F_none,
04630    STBI__F_sub,
04631    STBI__F_none,
04632    STBI__F_avg_first,
04633    STBI__F_paeth_first
04634 };
04635
04636 static int stbi__paeth(int a, int b, int c)
04637 {
04638    int p = a + b - c;
04639    int pa = abs(p-a);
04640    int pb = abs(p-b);
04641    int pc = abs(p-c);
04642    if (pa <= pb && pa <= pc) return a;
04643    if (pb <= pc) return b;
04644    return c;
04645 }
04646
04647 static const stbi_uc stbi__depth_scale_table[9] = { 0, 0xff, 0x55, 0, 0x11, 0,0,0, 0x01 };
04648
04649 // create the png data from post-deflated data
04650 static int stbi__create_png_image_raw(stbi__png *a, stbi_uc *raw, stbi__uint32 raw_len, int out_n,
      stbi__uint32 x, stbi__uint32 y, int depth, int color)
04651 {
04652    int bytes = (depth == 16? 2 : 1);
04653    stbi__context *s = a->s;
04654    stbi__uint32 i,j,stride = x*out_n*bytes;
04655    stbi__uint32 img_len, img_width_bytes;
04656    int k;
04657    int img_n = s->img_n; // copy it into a local for later
04658
04659    int output_bytes = out_n*bytes;
04660    int filter_bytes = img_n*bytes;
04661    int width = x;
04662
04663    STBI_ASSERT(out_n == s->img_n || out_n == s->img_n+1);
04664    a->out = (stbi_uc *) stbi__malloc_mad3(x, y, output_bytes, 0); // extra bytes to write off the end
      into
04665    if (!a->out) return stbi__err("outofmem", "Out of memory");
04666
04667    if (!stbi__mad3sizes_valid(img_n, x, depth, 7)) return stbi__err("too large", "Corrupt PNG");
04668    img_width_bytes = (((img_n * x * depth) + 7) >> 3);
04669    img_len = (img_width_bytes + 1) * y;
04670
04671    // we used to check for exact match between raw_len and img_len on non-interlaced PNGs,
04672    // but issue #276 reported a PNG in the wild that had extra data at the end (all zeros),
04673    // so just check for raw_len < img_len always.
04674    if (raw_len < img_len) return stbi__err("not enough pixels","Corrupt PNG");
04675
04676    for (j=0; j < y; ++j) {
04677       stbi_uc *cur = a->out + stride*j;
04678       stbi_uc *prior;
04679       int filter = *raw++;
04680
04681       if (filter > 4)
04682          return stbi__err("invalid filter","Corrupt PNG");
04683
04684       if (depth < 8) {
04685          if (img_width_bytes > x) return stbi__err("invalid width","Corrupt PNG");
04686          cur += x*out_n - img_width_bytes; // store output to the rightmost img_len bytes, so we can
      decode in place
04687          filter_bytes = 1;
04688          width = img_width_bytes;
04689       }
04690       prior = cur - stride; // bugfix: need to compute this after 'cur +=' computation above
04691
04692       // if first row, use special filter that doesn't sample previous row
04693       if (j == 0) filter = first_row_filter[filter];
04694
04695       // handle first byte explicitly
04696       for (k=0; k < filter_bytes; ++k) {
04697          switch (filter) {
04698             case STBI__F_none       : cur[k] = raw[k]; break;
04699             case STBI__F_sub        : cur[k] = raw[k]; break;
04700             case STBI__F_up         : cur[k] = STBI__BYTECAST(raw[k] + prior[k]); break;
04701             case STBI__F_avg        : cur[k] = STBI__BYTECAST(raw[k] + (prior[k]>>1)); break;
04702             case STBI__F_paeth      : cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(0,prior[k],0));
```

```
      break;
04703             case STBI__F_avg_first  : cur[k] = raw[k]; break;
04704             case STBI__F_paeth_first: cur[k] = raw[k]; break;
04705          }
04706       }
04707
04708       if (depth == 8) {
04709          if (img_n != out_n)
04710             cur[img_n] = 255; // first pixel
04711          raw += img_n;
04712          cur += out_n;
04713          prior += out_n;
04714       } else if (depth == 16) {
04715          if (img_n != out_n) {
04716             cur[filter_bytes]   = 255; // first pixel top byte
04717             cur[filter_bytes+1] = 255; // first pixel bottom byte
04718          }
04719          raw += filter_bytes;
04720          cur += output_bytes;
04721          prior += output_bytes;
04722       } else {
04723          raw += 1;
04724          cur += 1;
04725          prior += 1;
04726       }
04727
04728       // this is a little gross, so that we don't switch per-pixel or per-component
04729       if (depth < 8 || img_n == out_n) {
04730          int nk = (width - 1)*filter_bytes;
04731          #define STBI__CASE(f) \
04732              case f:     \
04733                 for (k=0; k < nk; ++k)
04734          switch (filter) {
04735             // "none" filter turns into a memcpy here; make that explicit.
04736             case STBI__F_none:         memcpy(cur, raw, nk); break;
04737             STBI__CASE(STBI__F_sub)          { cur[k] = STBI__BYTECAST(raw[k] + cur[k-filter_bytes]);
      } break;
04738             STBI__CASE(STBI__F_up)           { cur[k] = STBI__BYTECAST(raw[k] + prior[k]); } break;
04739             STBI__CASE(STBI__F_avg)          { cur[k] = STBI__BYTECAST(raw[k] + ((prior[k] +
      cur[k-filter_bytes])>>1)); } break;
04740             STBI__CASE(STBI__F_paeth)        { cur[k] = STBI__BYTECAST(raw[k] +
      stbi__paeth(cur[k-filter_bytes],prior[k],prior[k-filter_bytes])); } break;
04741             STBI__CASE(STBI__F_avg_first)    { cur[k] = STBI__BYTECAST(raw[k] + (cur[k-filter_bytes] >>
      1)); } break;
04742             STBI__CASE(STBI__F_paeth_first)  { cur[k] = STBI__BYTECAST(raw[k] +
      stbi__paeth(cur[k-filter_bytes],0,0)); } break;
04743          }
04744          #undef STBI__CASE
04745          raw += nk;
04746       } else {
04747          STBI_ASSERT(img_n+1 == out_n);
04748          #define STBI__CASE(f) \
04749              case f:     \
04750                 for (i=x-1; i >= 1; --i,
      cur[filter_bytes]=255,raw+=filter_bytes,cur+=output_bytes,prior+=output_bytes) \
04751                    for (k=0; k < filter_bytes; ++k)
04752          switch (filter) {
04753             STBI__CASE(STBI__F_none)         { cur[k] = raw[k]; } break;
04754             STBI__CASE(STBI__F_sub)          { cur[k] = STBI__BYTECAST(raw[k] + cur[k- output_bytes]);
      } break;
04755             STBI__CASE(STBI__F_up)           { cur[k] = STBI__BYTECAST(raw[k] + prior[k]); } break;
04756             STBI__CASE(STBI__F_avg)          { cur[k] = STBI__BYTECAST(raw[k] + ((prior[k] + cur[k-
      output_bytes])>>1)); } break;
04757             STBI__CASE(STBI__F_paeth)        { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k-
      output_bytes],prior[k],prior[k- output_bytes])); } break;
04758             STBI__CASE(STBI__F_avg_first)    { cur[k] = STBI__BYTECAST(raw[k] + (cur[k- output_bytes]
      >> 1)); } break;
04759             STBI__CASE(STBI__F_paeth_first)  { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k-
      output_bytes],0,0)); } break;
04760          }
04761          #undef STBI__CASE
04762
04763          // the loop above sets the high byte of the pixels' alpha, but for
04764          // 16 bit png files we also need the low byte set. we'll do that here.
04765          if (depth == 16) {
04766             cur = a->out + stride*j; // start at the beginning of the row again
04767             for (i=0; i < x; ++i,cur+=output_bytes) {
04768                cur[filter_bytes+1] = 255;
04769             }
04770          }
04771       }
04772    }
04773
04774    // we make a separate pass to expand bits to pixels; for performance,
04775    // this could run two scanlines behind the above code, so it won't
04776    // intefere with filtering but will still be in the cache.
04777    if (depth < 8) {
```

```
04778       for (j=0; j < y; ++j) {
04779          stbi_uc *cur = a->out + stride*j;
04780          stbi_uc *in  = a->out + stride*j + x*out_n - img_width_bytes;
04781          // unpack 1/2/4-bit into a 8-bit buffer. allows us to keep the common 8-bit path optimal at
    minimal cost for 1/2/4-bit
04782          // png guarante byte alignment, if width is not multiple of 8/4/2 we'll decode dummy trailing
    data that will be skipped in the later loop
04783          stbi_uc scale = (color == 0) ? stbi__depth_scale_table[depth] : 1; // scale grayscale values
    to 0..255 range
04784
04785          // note that the final byte might overshoot and write more data than desired.
04786          // we can allocate enough data that this never writes out of memory, but it
04787          // could also overwrite the next scanline. can it overwrite non-empty data
04788          // on the next scanline? yes, consider 1-pixel-wide scanlines with 1-bit-per-pixel.
04789          // so we need to explicitly clamp the final ones
04790
04791          if (depth == 4) {
04792             for (k=x*img_n; k >= 2; k-=2, ++in) {
04793                *cur++ = scale * ((*in » 4)       );
04794                *cur++ = scale * ((*in     ) & 0x0f);
04795             }
04796             if (k > 0) *cur++ = scale * ((*in » 4)       );
04797          } else if (depth == 2) {
04798             for (k=x*img_n; k >= 4; k-=4, ++in) {
04799                *cur++ = scale * ((*in » 6)       );
04800                *cur++ = scale * ((*in » 4) & 0x03);
04801                *cur++ = scale * ((*in » 2) & 0x03);
04802                *cur++ = scale * ((*in     ) & 0x03);
04803             }
04804             if (k > 0) *cur++ = scale * ((*in » 6)       );
04805             if (k > 1) *cur++ = scale * ((*in » 4) & 0x03);
04806             if (k > 2) *cur++ = scale * ((*in » 2) & 0x03);
04807          } else if (depth == 1) {
04808             for (k=x*img_n; k >= 8; k-=8, ++in) {
04809                *cur++ = scale * ((*in » 7)       );
04810                *cur++ = scale * ((*in » 6) & 0x01);
04811                *cur++ = scale * ((*in » 5) & 0x01);
04812                *cur++ = scale * ((*in » 4) & 0x01);
04813                *cur++ = scale * ((*in » 3) & 0x01);
04814                *cur++ = scale * ((*in » 2) & 0x01);
04815                *cur++ = scale * ((*in » 1) & 0x01);
04816                *cur++ = scale * ((*in     ) & 0x01);
04817             }
04818             if (k > 0) *cur++ = scale * ((*in » 7)       );
04819             if (k > 1) *cur++ = scale * ((*in » 6) & 0x01);
04820             if (k > 2) *cur++ = scale * ((*in » 5) & 0x01);
04821             if (k > 3) *cur++ = scale * ((*in » 4) & 0x01);
04822             if (k > 4) *cur++ = scale * ((*in » 3) & 0x01);
04823             if (k > 5) *cur++ = scale * ((*in » 2) & 0x01);
04824             if (k > 6) *cur++ = scale * ((*in » 1) & 0x01);
04825          }
04826          if (img_n != out_n) {
04827             int q;
04828             // insert alpha = 255
04829             cur = a->out + stride*j;
04830             if (img_n == 1) {
04831                for (q=x-1; q >= 0; --q) {
04832                   cur[q*2+1] = 255;
04833                   cur[q*2+0] = cur[q];
04834                }
04835             } else {
04836                STBI_ASSERT(img_n == 3);
04837                for (q=x-1; q >= 0; --q) {
04838                   cur[q*4+3] = 255;
04839                   cur[q*4+2] = cur[q*3+2];
04840                   cur[q*4+1] = cur[q*3+1];
04841                   cur[q*4+0] = cur[q*3+0];
04842                }
04843             }
04844          }
04845       }
04846    } else if (depth == 16) {
04847       // force the image data from big-endian to platform-native.
04848       // this is done in a separate pass due to the decoding relying
04849       // on the data being untouched, but could probably be done
04850       // per-line during decode if care is taken.
04851       stbi_uc *cur = a->out;
04852       stbi__uint16 *cur16 = (stbi__uint16*)cur;
04853
04854       for(i=0; i < x*y*out_n; ++i,cur16++,cur+=2) {
04855          *cur16 = (cur[0] « 8) | cur[1];
04856       }
04857    }
04858
04859    return 1;
04860 }
04861
```

```
04862 static int stbi__create_png_image(stbi__png *a, stbi_uc *image_data, stbi__uint32 image_data_len, int
      out_n, int depth, int color, int interlaced)
04863 {
04864    int bytes = (depth == 16 ? 2 : 1);
04865    int out_bytes = out_n * bytes;
04866    stbi_uc *final;
04867    int p;
04868    if (!interlaced)
04869       return stbi__create_png_image_raw(a, image_data, image_data_len, out_n, a->s->img_x,
      a->s->img_y, depth, color);
04870
04871    // de-interlacing
04872    final = (stbi_uc *) stbi__malloc_mad3(a->s->img_x, a->s->img_y, out_bytes, 0);
04873    if (!final) return stbi__err("outofmem", "Out of memory");
04874    for (p=0; p < 7; ++p) {
04875       int xorig[] = { 0,4,0,2,0,1,0 };
04876       int yorig[] = { 0,0,4,0,2,0,1 };
04877       int xspc[]  = { 8,8,4,4,2,2,1 };
04878       int yspc[]  = { 8,8,8,4,4,2,2 };
04879       int i,j,x,y;
04880       // pass1_x[4] = 0, pass1_x[5] = 1, pass1_x[12] = 1
04881       x = (a->s->img_x - xorig[p] + xspc[p]-1) / xspc[p];
04882       y = (a->s->img_y - yorig[p] + yspc[p]-1) / yspc[p];
04883       if (x && y) {
04884          stbi__uint32 img_len = ((((a->s->img_n * x * depth) + 7) >> 3) + 1) * y;
04885          if (!stbi__create_png_image_raw(a, image_data, image_data_len, out_n, x, y, depth, color)) {
04886             STBI_FREE(final);
04887             return 0;
04888          }
04889          for (j=0; j < y; ++j) {
04890             for (i=0; i < x; ++i) {
04891                int out_y = j*yspc[p]+yorig[p];
04892                int out_x = i*xspc[p]+xorig[p];
04893                memcpy(final + out_y*a->s->img_x*out_bytes + out_x*out_bytes,
04894                       a->out + (j*x+i)*out_bytes, out_bytes);
04895             }
04896          }
04897          STBI_FREE(a->out);
04898          image_data += img_len;
04899          image_data_len -= img_len;
04900       }
04901    }
04902    a->out = final;
04903
04904    return 1;
04905 }
04906
04907 static int stbi__compute_transparency(stbi__png *z, stbi_uc tc[3], int out_n)
04908 {
04909    stbi__context *s = z->s;
04910    stbi__uint32 i, pixel_count = s->img_x * s->img_y;
04911    stbi_uc *p = z->out;
04912
04913    // compute color-based transparency, assuming we've
04914    // already got 255 as the alpha value in the output
04915    STBI_ASSERT(out_n == 2 || out_n == 4);
04916
04917    if (out_n == 2) {
04918       for (i=0; i < pixel_count; ++i) {
04919          p[1] = (p[0] == tc[0] ? 0 : 255);
04920          p += 2;
04921       }
04922    } else {
04923       for (i=0; i < pixel_count; ++i) {
04924          if (p[0] == tc[0] && p[1] == tc[1] && p[2] == tc[2])
04925             p[3] = 0;
04926          p += 4;
04927       }
04928    }
04929    return 1;
04930 }
04931
04932 static int stbi__compute_transparency16(stbi__png *z, stbi__uint16 tc[3], int out_n)
04933 {
04934    stbi__context *s = z->s;
04935    stbi__uint32 i, pixel_count = s->img_x * s->img_y;
04936    stbi__uint16 *p = (stbi__uint16*) z->out;
04937
04938    // compute color-based transparency, assuming we've
04939    // already got 65535 as the alpha value in the output
04940    STBI_ASSERT(out_n == 2 || out_n == 4);
04941
04942    if (out_n == 2) {
04943       for (i = 0; i < pixel_count; ++i) {
04944          p[1] = (p[0] == tc[0] ? 0 : 65535);
04945          p += 2;
04946       }
```

```
04947    } else {
04948       for (i = 0; i < pixel_count; ++i) {
04949          if (p[0] == tc[0] && p[1] == tc[1] && p[2] == tc[2])
04950             p[3] = 0;
04951          p += 4;
04952       }
04953    }
04954    return 1;
04955 }
04956
04957 static int stbi__expand_png_palette(stbi__png *a, stbi_uc *palette, int len, int pal_img_n)
04958 {
04959    stbi__uint32 i, pixel_count = a->s->img_x * a->s->img_y;
04960    stbi_uc *p, *temp_out, *orig = a->out;
04961
04962    p = (stbi_uc *) stbi__malloc_mad2(pixel_count, pal_img_n, 0);
04963    if (p == NULL) return stbi__err("outofmem", "Out of memory");
04964
04965    // between here and free(out) below, exitting would leak
04966    temp_out = p;
04967
04968    if (pal_img_n == 3) {
04969       for (i=0; i < pixel_count; ++i) {
04970          int n = orig[i]*4;
04971          p[0] = palette[n  ];
04972          p[1] = palette[n+1];
04973          p[2] = palette[n+2];
04974          p += 3;
04975       }
04976    } else {
04977       for (i=0; i < pixel_count; ++i) {
04978          int n = orig[i]*4;
04979          p[0] = palette[n  ];
04980          p[1] = palette[n+1];
04981          p[2] = palette[n+2];
04982          p[3] = palette[n+3];
04983          p += 4;
04984       }
04985    }
04986    STBI_FREE(a->out);
04987    a->out = temp_out;
04988
04989    STBI_NOTUSED(len);
04990
04991    return 1;
04992 }
04993
04994 static int stbi__unpremultiply_on_load_global = 0;
04995 static int stbi__de_iphone_flag_global = 0;
04996
04997 STBIDEF void stbi_set_unpremultiply_on_load(int flag_true_if_should_unpremultiply)
04998 {
04999    stbi__unpremultiply_on_load_global = flag_true_if_should_unpremultiply;
05000 }
05001
05002 STBIDEF void stbi_convert_iphone_png_to_rgb(int flag_true_if_should_convert)
05003 {
05004    stbi__de_iphone_flag_global = flag_true_if_should_convert;
05005 }
05006
05007 #ifndef STBI_THREAD_LOCAL
05008 #define stbi__unpremultiply_on_load  stbi__unpremultiply_on_load_global
05009 #define stbi__de_iphone_flag  stbi__de_iphone_flag_global
05010 #else
05011 static STBI_THREAD_LOCAL int stbi__unpremultiply_on_load_local, stbi__unpremultiply_on_load_set;
05012 static STBI_THREAD_LOCAL int stbi__de_iphone_flag_local, stbi__de_iphone_flag_set;
05013
05014 STBIDEF void stbi_set_unpremultiply_on_load_thread(int flag_true_if_should_unpremultiply)
05015 {
05016    stbi__unpremultiply_on_load_local = flag_true_if_should_unpremultiply;
05017    stbi__unpremultiply_on_load_set = 1;
05018 }
05019
05020 STBIDEF void stbi_convert_iphone_png_to_rgb_thread(int flag_true_if_should_convert)
05021 {
05022    stbi__de_iphone_flag_local = flag_true_if_should_convert;
05023    stbi__de_iphone_flag_set = 1;
05024 }
05025
05026 #define stbi__unpremultiply_on_load  (stbi__unpremultiply_on_load_set           \
05027                                       ? stbi__unpremultiply_on_load_local      \
05028                                       : stbi__unpremultiply_on_load_global)
05029 #define stbi__de_iphone_flag  (stbi__de_iphone_flag_set                         \
05030                                ? stbi__de_iphone_flag_local                    \
05031                                : stbi__de_iphone_flag_global)
05032 #endif // STBI_THREAD_LOCAL
05033
```

```
05034 static void stbi__de_iphone(stbi__png *z)
05035 {
05036    stbi__context *s = z->s;
05037    stbi__uint32 i, pixel_count = s->img_x * s->img_y;
05038    stbi_uc *p = z->out;
05039
05040    if (s->img_out_n == 3) {  // convert bgr to rgb
05041       for (i=0; i < pixel_count; ++i) {
05042          stbi_uc t = p[0];
05043          p[0] = p[2];
05044          p[2] = t;
05045          p += 3;
05046       }
05047    } else {
05048       STBI_ASSERT(s->img_out_n == 4);
05049       if (stbi__unpremultiply_on_load) {
05050          // convert bgr to rgb and unpremultiply
05051          for (i=0; i < pixel_count; ++i) {
05052             stbi_uc a = p[3];
05053             stbi_uc t = p[0];
05054             if (a) {
05055                stbi_uc half = a / 2;
05056                p[0] = (p[2] * 255 + half) / a;
05057                p[1] = (p[1] * 255 + half) / a;
05058                p[2] = ( t   * 255 + half) / a;
05059             } else {
05060                p[0] = p[2];
05061                p[2] = t;
05062             }
05063             p += 4;
05064          }
05065       } else {
05066          // convert bgr to rgb
05067          for (i=0; i < pixel_count; ++i) {
05068             stbi_uc t = p[0];
05069             p[0] = p[2];
05070             p[2] = t;
05071             p += 4;
05072          }
05073       }
05074    }
05075 }
05076
05077 #define STBI__PNG_TYPE(a,b,c,d)  (((unsigned) (a) « 24) + ((unsigned) (b) « 16) + ((unsigned) (c) « 8)
       + (unsigned) (d))
05078
05079 static int stbi__parse_png_file(stbi__png *z, int scan, int req_comp)
05080 {
05081    stbi_uc palette[1024], pal_img_n=0;
05082    stbi_uc has_trans=0, tc[3]={0};
05083    stbi__uint16 tc16[3];
05084    stbi__uint32 ioff=0, idata_limit=0, i, pal_len=0;
05085    int first=1,k,interlace=0, color=0, is_iphone=0;
05086    stbi__context *s = z->s;
05087
05088    z->expanded = NULL;
05089    z->idata = NULL;
05090    z->out = NULL;
05091
05092    if (!stbi__check_png_header(s)) return 0;
05093
05094    if (scan == STBI__SCAN_type) return 1;
05095
05096    for (;;) {
05097       stbi__pngchunk c = stbi__get_chunk_header(s);
05098       switch (c.type) {
05099          case STBI__PNG_TYPE('C','g','B','I'):
05100             is_iphone = 1;
05101             stbi__skip(s, c.length);
05102             break;
05103          case STBI__PNG_TYPE('I','H','D','R'): {
05104             int comp,filter;
05105             if (!first) return stbi__err("multiple IHDR","Corrupt PNG");
05106             first = 0;
05107             if (c.length != 13) return stbi__err("bad IHDR len","Corrupt PNG");
05108             s->img_x = stbi__get32be(s);
05109             s->img_y = stbi__get32be(s);
05110             if (s->img_y > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image
       (corrupt?)");
05111             if (s->img_x > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image
       (corrupt?)");
05112             z->depth = stbi__get8(s);  if (z->depth != 1 && z->depth != 2 && z->depth != 4 && z->depth
       != 8 && z->depth != 16)  return stbi__err("1/2/4/8/16-bit only","PNG not supported: 1/2/4/8/16-bit
       only");
05113             color = stbi__get8(s);  if (color > 6)         return stbi__err("bad ctype","Corrupt
       PNG");
05114             if (color == 3 && z->depth == 16)                  return stbi__err("bad ctype","Corrupt
```

```
PNG");
05115             if (color == 3) pal_img_n = 3; else if (color & 1) return stbi__err("bad ctype","Corrupt
PNG");
05116             comp  = stbi__get8(s);  if (comp) return stbi__err("bad comp method","Corrupt PNG");
05117             filter= stbi__get8(s);  if (filter) return stbi__err("bad filter method","Corrupt PNG");
05118             interlace = stbi__get8(s); if (interlace>1) return stbi__err("bad interlace
method","Corrupt PNG");
05119             if (!s->img_x || !s->img_y) return stbi__err("0-pixel image","Corrupt PNG");
05120             if (!pal_img_n) {
05121                s->img_n = (color & 2 ? 3 : 1) + (color & 4 ? 1 : 0);
05122                if ((1 << 30) / s->img_x / s->img_n < s->img_y) return stbi__err("too large", "Image too
large to decode");
05123             } else {
05124                // if paletted, then pal_n is our final components, and
05125                // img_n is # components to decompress/filter.
05126                s->img_n = 1;
05127                if ((1 << 30) / s->img_x / 4 < s->img_y) return stbi__err("too large","Corrupt PNG");
05128             }
05129             // even with SCAN_header, have to scan to see if we have a tRNS
05130             break;
05131          }
05132
05133          case STBI__PNG_TYPE('P','L','T','E'):  {
05134             if (first) return stbi__err("first not IHDR", "Corrupt PNG");
05135             if (c.length > 256*3) return stbi__err("invalid PLTE","Corrupt PNG");
05136             pal_len = c.length / 3;
05137             if (pal_len * 3 != c.length) return stbi__err("invalid PLTE","Corrupt PNG");
05138             for (i=0; i < pal_len; ++i) {
05139                palette[i*4+0] = stbi__get8(s);
05140                palette[i*4+1] = stbi__get8(s);
05141                palette[i*4+2] = stbi__get8(s);
05142                palette[i*4+3] = 255;
05143             }
05144             break;
05145          }
05146
05147          case STBI__PNG_TYPE('t','R','N','S'): {
05148             if (first) return stbi__err("first not IHDR", "Corrupt PNG");
05149             if (z->idata) return stbi__err("tRNS after IDAT","Corrupt PNG");
05150             if (pal_img_n) {
05151                if (scan == STBI__SCAN_header) { s->img_n = 4; return 1; }
05152                if (pal_len == 0) return stbi__err("tRNS before PLTE","Corrupt PNG");
05153                if (c.length > pal_len) return stbi__err("bad tRNS len","Corrupt PNG");
05154                pal_img_n = 4;
05155                for (i=0; i < c.length; ++i)
05156                   palette[i*4+3] = stbi__get8(s);
05157             } else {
05158                if (!(s->img_n & 1)) return stbi__err("tRNS with alpha","Corrupt PNG");
05159                if (c.length != (stbi__uint32) s->img_n*2) return stbi__err("bad tRNS len","Corrupt
PNG");
05160                has_trans = 1;
05161                // non-paletted with tRNS = constant alpha. if header-scanning, we can stop now.
05162                if (scan == STBI__SCAN_header) { ++s->img_n; return 1; }
05163                if (z->depth == 16) {
05164                   for (k = 0; k < s->img_n; ++k) tc16[k] = (stbi__uint16)stbi__get16be(s); // copy the
values as-is
05165                } else {
05166                   for (k = 0; k < s->img_n; ++k) tc[k] = (stbi_uc)(stbi__get16be(s) & 255) *
stbi__depth_scale_table[z->depth]; // non 8-bit images will be larger
05167                }
05168             }
05169             break;
05170          }
05171
05172          case STBI__PNG_TYPE('I','D','A','T'): {
05173             if (first) return stbi__err("first not IHDR", "Corrupt PNG");
05174             if (pal_img_n && !pal_len) return stbi__err("no PLTE","Corrupt PNG");
05175             if (scan == STBI__SCAN_header) {
05176                // header scan definitely stops at first IDAT
05177                if (pal_img_n)
05178                   s->img_n = pal_img_n;
05179                return 1;
05180             }
05181             if (c.length > (1u << 30)) return stbi__err("IDAT size limit", "IDAT section larger than
2^30 bytes");
05182             if ((int)(ioff + c.length) < (int)ioff) return 0;
05183             if (ioff + c.length > idata_limit) {
05184                stbi__uint32 idata_limit_old = idata_limit;
05185                stbi_uc *p;
05186                if (idata_limit == 0) idata_limit = c.length > 4096 ? c.length : 4096;
05187                while (ioff + c.length > idata_limit)
05188                   idata_limit *= 2;
05189                STBI_NOTUSED(idata_limit_old);
05190                p = (stbi_uc *) STBI_REALLOC_SIZED(z->idata, idata_limit_old, idata_limit); if (p ==
NULL) return stbi__err("outofmem", "Out of memory");
05191                z->idata = p;
05192             }
```

```
05193             if (!stbi__getn(s, z->idata+ioff,c.length)) return stbi__err("outofdata","Corrupt PNG");
05194             ioff += c.length;
05195             break;
05196          }
05197
05198          case STBI__PNG_TYPE('I','E','N','D'): {
05199             stbi__uint32 raw_len, bpl;
05200             if (first) return stbi__err("first not IHDR", "Corrupt PNG");
05201             if (scan != STBI__SCAN_load) return 1;
05202             if (z->idata == NULL) return stbi__err("no IDAT","Corrupt PNG");
05203             // initial guess for decoded data size to avoid unnecessary reallocs
05204             bpl = (s->img_x * z->depth + 7) / 8; // bytes per line, per component
05205             raw_len = bpl * s->img_y * s->img_n /* pixels */ + s->img_y /* filter mode per row */;
05206             z->expanded = (stbi_uc *) stbi_zlib_decode_malloc_guesssize_headerflag((char *) z->idata,
    ioff, raw_len, (int *) &raw_len, !is_iphone);
05207             if (z->expanded == NULL) return 0; // zlib should set error
05208             STBI_FREE(z->idata); z->idata = NULL;
05209             if ((req_comp == s->img_n+1 && req_comp != 3 && !pal_img_n) || has_trans)
05210                s->img_out_n = s->img_n+1;
05211             else
05212                s->img_out_n = s->img_n;
05213             if (!stbi__create_png_image(z, z->expanded, raw_len, s->img_out_n, z->depth, color,
    interlace)) return 0;
05214             if (has_trans) {
05215                if (z->depth == 16) {
05216                   if (!stbi__compute_transparency16(z, tc16, s->img_out_n)) return 0;
05217                } else {
05218                   if (!stbi__compute_transparency(z, tc, s->img_out_n)) return 0;
05219                }
05220             }
05221             if (is_iphone && stbi__de_iphone_flag && s->img_out_n > 2)
05222                stbi__de_iphone(z);
05223             if (pal_img_n) {
05224                // pal_img_n == 3 or 4
05225                s->img_n = pal_img_n; // record the actual colors we had
05226                s->img_out_n = pal_img_n;
05227                if (req_comp >= 3) s->img_out_n = req_comp;
05228                if (!stbi__expand_png_palette(z, palette, pal_len, s->img_out_n))
05229                   return 0;
05230             } else if (has_trans) {
05231                // non-paletted image with tRNS -> source image has (constant) alpha
05232                ++s->img_n;
05233             }
05234             STBI_FREE(z->expanded); z->expanded = NULL;
05235             // end of PNG chunk, read and skip CRC
05236             stbi__get32be(s);
05237             return 1;
05238          }
05239
05240          default:
05241             // if critical, fail
05242             if (first) return stbi__err("first not IHDR", "Corrupt PNG");
05243             if ((c.type & (1 << 29)) == 0) {
05244                #ifndef STBI_NO_FAILURE_STRINGS
05245                // not threadsafe
05246                static char invalid_chunk[] = "XXXX PNG chunk not known";
05247                invalid_chunk[0] = STBI__BYTECAST(c.type >> 24);
05248                invalid_chunk[1] = STBI__BYTECAST(c.type >> 16);
05249                invalid_chunk[2] = STBI__BYTECAST(c.type >>  8);
05250                invalid_chunk[3] = STBI__BYTECAST(c.type >>  0);
05251                #endif
05252                return stbi__err(invalid_chunk, "PNG not supported: unknown PNG chunk type");
05253             }
05254             stbi__skip(s, c.length);
05255             break;
05256       }
05257       // end of PNG chunk, read and skip CRC
05258       stbi__get32be(s);
05259    }
05260 }
05261
05262 static void *stbi__do_png(stbi__png *p, int *x, int *y, int *n, int req_comp, stbi__result_info *ri)
05263 {
05264    void *result=NULL;
05265    if (req_comp < 0 || req_comp > 4) return stbi__errpuc("bad req_comp", "Internal error");
05266    if (stbi__parse_png_file(p, STBI__SCAN_load, req_comp)) {
05267       if (p->depth <= 8)
05268          ri->bits_per_channel = 8;
05269       else if (p->depth == 16)
05270          ri->bits_per_channel = 16;
05271       else
05272          return stbi__errpuc("bad bits_per_channel", "PNG not supported: unsupported color depth");
05273       result = p->out;
05274       p->out = NULL;
05275       if (req_comp && req_comp != p->s->img_out_n) {
05276          if (ri->bits_per_channel == 8)
05277             result = stbi__convert_format((unsigned char *) result, p->s->img_out_n, req_comp,
```

```
      p->s->img_x, p->s->img_y);
05278          else
05279             result = stbi__convert_format16((stbi__uint16 *) result, p->s->img_out_n, req_comp,
      p->s->img_x, p->s->img_y);
05280          p->s->img_out_n = req_comp;
05281          if (result == NULL) return result;
05282       }
05283       *x = p->s->img_x;
05284       *y = p->s->img_y;
05285       if (n) *n = p->s->img_n;
05286    }
05287    STBI_FREE(p->out);      p->out      = NULL;
05288    STBI_FREE(p->expanded); p->expanded = NULL;
05289    STBI_FREE(p->idata);    p->idata    = NULL;
05290
05291    return result;
05292 }
05293
05294 static void *stbi__png_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri)
05295 {
05296    stbi__png p;
05297    p.s = s;
05298    return stbi__do_png(&p, x,y,comp,req_comp, ri);
05299 }
05300
05301 static int stbi__png_test(stbi__context *s)
05302 {
05303    int r;
05304    r = stbi__check_png_header(s);
05305    stbi__rewind(s);
05306    return r;
05307 }
05308
05309 static int stbi__png_info_raw(stbi__png *p, int *x, int *y, int *comp)
05310 {
05311    if (!stbi__parse_png_file(p, STBI__SCAN_header, 0)) {
05312       stbi__rewind( p->s );
05313       return 0;
05314    }
05315    if (x) *x = p->s->img_x;
05316    if (y) *y = p->s->img_y;
05317    if (comp) *comp = p->s->img_n;
05318    return 1;
05319 }
05320
05321 static int stbi__png_info(stbi__context *s, int *x, int *y, int *comp)
05322 {
05323    stbi__png p;
05324    p.s = s;
05325    return stbi__png_info_raw(&p, x, y, comp);
05326 }
05327
05328 static int stbi__png_is16(stbi__context *s)
05329 {
05330    stbi__png p;
05331    p.s = s;
05332    if (!stbi__png_info_raw(&p, NULL, NULL, NULL))
05333        return 0;
05334    if (p.depth != 16) {
05335       stbi__rewind(p.s);
05336       return 0;
05337    }
05338    return 1;
05339 }
05340 #endif
05341
05342 // Microsoft/Windows BMP image
05343
05344 #ifndef STBI_NO_BMP
05345 static int stbi__bmp_test_raw(stbi__context *s)
05346 {
05347    int r;
05348    int sz;
05349    if (stbi__get8(s) != 'B') return 0;
05350    if (stbi__get8(s) != 'M') return 0;
05351    stbi__get32le(s); // discard filesize
05352    stbi__get16le(s); // discard reserved
05353    stbi__get16le(s); // discard reserved
05354    stbi__get32le(s); // discard data offset
05355    sz = stbi__get32le(s);
05356    r = (sz == 12 || sz == 40 || sz == 56 || sz == 108 || sz == 124);
05357    return r;
05358 }
05359
05360 static int stbi__bmp_test(stbi__context *s)
05361 {
```

```
05362    int r = stbi__bmp_test_raw(s);
05363    stbi__rewind(s);
05364    return r;
05365 }
05366
05367
05368 // returns 0..31 for the highest set bit
05369 static int stbi__high_bit(unsigned int z)
05370 {
05371    int n=0;
05372    if (z == 0) return -1;
05373    if (z >= 0x10000) { n += 16; z »= 16; }
05374    if (z >= 0x00100) { n +=  8; z »=  8; }
05375    if (z >= 0x00010) { n +=  4; z »=  4; }
05376    if (z >= 0x00004) { n +=  2; z »=  2; }
05377    if (z >= 0x00002) { n +=  1;/* »=  1;*/ }
05378    return n;
05379 }
05380
05381 static int stbi__bitcount(unsigned int a)
05382 {
05383    a = (a & 0x55555555) + ((a »  1) & 0x55555555); // max 2
05384    a = (a & 0x33333333) + ((a »  2) & 0x33333333); // max 4
05385    a = (a + (a » 4)) & 0x0f0f0f0f; // max 8 per 4, now 8 bits
05386    a = (a + (a » 8)); // max 16 per 8 bits
05387    a = (a + (a » 16)); // max 32 per 8 bits
05388    return a & 0xff;
05389 }
05390
05391 // extract an arbitrarily-aligned N-bit value (N=bits)
05392 // from v, and then make it 8-bits long and fractionally
05393 // extend it to full full range.
05394 static int stbi__shiftsigned(unsigned int v, int shift, int bits)
05395 {
05396    static unsigned int mul_table[9] = {
05397       0,
05398       0xff/*0b11111111*/, 0x55/*0b01010101*/, 0x49/*0b01001001*/, 0x11/*0b00010001*/,
05399       0x21/*0b00100001*/, 0x41/*0b01000001*/, 0x81/*0b10000001*/, 0x01/*0b00000001*/,
05400    };
05401    static unsigned int shift_table[9] = {
05402       0, 0,0,1,0,2,4,6,0,
05403    };
05404    if (shift < 0)
05405       v «= -shift;
05406    else
05407       v »= shift;
05408    STBI_ASSERT(v < 256);
05409    v »= (8-bits);
05410    STBI_ASSERT(bits >= 0 && bits <= 8);
05411    return (int) ((unsigned) v * mul_table[bits]) » shift_table[bits];
05412 }
05413
05414 typedef struct
05415 {
05416    int bpp, offset, hsz;
05417    unsigned int mr,mg,mb,ma, all_a;
05418    int extra_read;
05419 } stbi__bmp_data;
05420
05421 static int stbi__bmp_set_mask_defaults(stbi__bmp_data *info, int compress)
05422 {
05423    // BI_BITFIELDS specifies masks explicitly, don't override
05424    if (compress == 3)
05425       return 1;
05426
05427    if (compress == 0) {
05428       if (info->bpp == 16) {
05429          info->mr = 31u « 10;
05430          info->mg = 31u «  5;
05431          info->mb = 31u «  0;
05432       } else if (info->bpp == 32) {
05433          info->mr = 0xffu « 16;
05434          info->mg = 0xffu «  8;
05435          info->mb = 0xffu «  0;
05436          info->ma = 0xffu « 24;
05437          info->all_a = 0; // if all_a is 0 at end, then we loaded alpha channel but it was all 0
05438       } else {
05439          // otherwise, use defaults, which is all-0
05440          info->mr = info->mg = info->mb = info->ma = 0;
05441       }
05442       return 1;
05443    }
05444    return 0; // error
05445 }
05446
05447 static void *stbi__bmp_parse_header(stbi__context *s, stbi__bmp_data *info)
05448 {
```

```
05449    int hsz;
05450    if (stbi__get8(s) != 'B' || stbi__get8(s) != 'M') return stbi__errpuc("not BMP", "Corrupt BMP");
05451    stbi__get32le(s); // discard filesize
05452    stbi__get16le(s); // discard reserved
05453    stbi__get16le(s); // discard reserved
05454    info->offset = stbi__get32le(s);
05455    info->hsz = hsz = stbi__get32le(s);
05456    info->mr = info->mg = info->mb = info->ma = 0;
05457    info->extra_read = 14;
05458
05459    if (info->offset < 0) return stbi__errpuc("bad BMP", "bad BMP");
05460
05461    if (hsz != 12 && hsz != 40 && hsz != 56 && hsz != 108 && hsz != 124) return stbi__errpuc("unknown
       BMP", "BMP type not supported: unknown");
05462    if (hsz == 12) {
05463       s->img_x = stbi__get16le(s);
05464       s->img_y = stbi__get16le(s);
05465    } else {
05466       s->img_x = stbi__get32le(s);
05467       s->img_y = stbi__get32le(s);
05468    }
05469    if (stbi__get16le(s) != 1) return stbi__errpuc("bad BMP", "bad BMP");
05470    info->bpp = stbi__get16le(s);
05471    if (hsz != 12) {
05472       int compress = stbi__get32le(s);
05473       if (compress == 1 || compress == 2) return stbi__errpuc("BMP RLE", "BMP type not supported:
       RLE");
05474       if (compress >= 4) return stbi__errpuc("BMP JPEG/PNG", "BMP type not supported: unsupported
       compression"); // this includes PNG/JPEG modes
05475       if (compress == 3 && info->bpp != 16 && info->bpp != 32) return stbi__errpuc("bad BMP", "bad
       BMP"); // bitfields requires 16 or 32 bits/pixel
05476       stbi__get32le(s); // discard sizeof
05477       stbi__get32le(s); // discard hres
05478       stbi__get32le(s); // discard vres
05479       stbi__get32le(s); // discard colorsused
05480       stbi__get32le(s); // discard max important
05481       if (hsz == 40 || hsz == 56) {
05482          if (hsz == 56) {
05483             stbi__get32le(s);
05484             stbi__get32le(s);
05485             stbi__get32le(s);
05486             stbi__get32le(s);
05487          }
05488          if (info->bpp == 16 || info->bpp == 32) {
05489             if (compress == 0) {
05490                stbi__bmp_set_mask_defaults(info, compress);
05491             } else if (compress == 3) {
05492                info->mr = stbi__get32le(s);
05493                info->mg = stbi__get32le(s);
05494                info->mb = stbi__get32le(s);
05495                info->extra_read += 12;
05496                // not documented, but generated by photoshop and handled by mspaint
05497                if (info->mr == info->mg && info->mg == info->mb) {
05498                   // ?!?!?
05499                   return stbi__errpuc("bad BMP", "bad BMP");
05500                }
05501             } else
05502                return stbi__errpuc("bad BMP", "bad BMP");
05503          }
05504       } else {
05505          // V4/V5 header
05506          int i;
05507          if (hsz != 108 && hsz != 124)
05508             return stbi__errpuc("bad BMP", "bad BMP");
05509          info->mr = stbi__get32le(s);
05510          info->mg = stbi__get32le(s);
05511          info->mb = stbi__get32le(s);
05512          info->ma = stbi__get32le(s);
05513          if (compress != 3) // override mr/mg/mb unless in BI_BITFIELDS mode, as per docs
05514             stbi__bmp_set_mask_defaults(info, compress);
05515          stbi__get32le(s); // discard color space
05516          for (i=0; i < 12; ++i)
05517             stbi__get32le(s); // discard color space parameters
05518          if (hsz == 124) {
05519             stbi__get32le(s); // discard rendering intent
05520             stbi__get32le(s); // discard offset of profile data
05521             stbi__get32le(s); // discard size of profile data
05522             stbi__get32le(s); // discard reserved
05523          }
05524       }
05525    }
05526    return (void *) 1;
05527 }
05528
05529
05530 static void *stbi__bmp_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
       stbi__result_info *ri)
```

```
05531 {
05532    stbi_uc *out;
05533    unsigned int mr=0,mg=0,mb=0,ma=0, all_a;
05534    stbi_uc pal[256][4];
05535    int psize=0,i,j,width;
05536    int flip_vertically, pad, target;
05537    stbi__bmp_data info;
05538    STBI_NOTUSED(ri);
05539
05540    info.all_a = 255;
05541    if (stbi__bmp_parse_header(s, &info) == NULL)
05542       return NULL; // error code already set
05543
05544    flip_vertically = ((int) s->img_y) > 0;
05545    s->img_y = abs((int) s->img_y);
05546
05547    if (s->img_y > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
05548    if (s->img_x > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
05549
05550    mr = info.mr;
05551    mg = info.mg;
05552    mb = info.mb;
05553    ma = info.ma;
05554    all_a = info.all_a;
05555
05556    if (info.hsz == 12) {
05557       if (info.bpp < 24)
05558          psize = (info.offset - info.extra_read - 24) / 3;
05559    } else {
05560       if (info.bpp < 16)
05561          psize = (info.offset - info.extra_read - info.hsz) >> 2;
05562    }
05563    if (psize == 0) {
05564       // accept some number of extra bytes after the header, but if the offset points either to before
05565       // the header ends or implies a large amount of extra data, reject the file as malformed
05566       int bytes_read_so_far = s->callback_already_read + (int)(s->img_buffer -
     s->img_buffer_original);
05567       int header_limit = 1024; // max we actually read is below 256 bytes currently.
05568       int extra_data_limit = 256*4; // what ordinarily goes here is a palette; 256 entries*4 bytes is
     its max size.
05569       if (bytes_read_so_far <= 0 || bytes_read_so_far > header_limit) {
05570          return stbi__errpuc("bad header", "Corrupt BMP");
05571       }
05572       // we established that bytes_read_so_far is positive and sensible.
05573       // the first half of this test rejects offsets that are either too small positives, or
05574       // negative, and guarantees that info.offset >= bytes_read_so_far > 0. this in turn
05575       // ensures the number computed in the second half of the test can't overflow.
05576       if (info.offset < bytes_read_so_far || info.offset - bytes_read_so_far > extra_data_limit) {
05577          return stbi__errpuc("bad offset", "Corrupt BMP");
05578       } else {
05579          stbi__skip(s, info.offset - bytes_read_so_far);
05580       }
05581    }
05582
05583    if (info.bpp == 24 && ma == 0xff000000)
05584       s->img_n = 3;
05585    else
05586       s->img_n = ma ? 4 : 3;
05587    if (req_comp && req_comp >= 3) // we can directly decode 3 or 4
05588       target = req_comp;
05589    else
05590       target = s->img_n; // if they want monochrome, we'll post-convert
05591
05592    // sanity-check size
05593    if (!stbi__mad3sizes_valid(target, s->img_x, s->img_y, 0))
05594       return stbi__errpuc("too large", "Corrupt BMP");
05595
05596    out = (stbi_uc *) stbi__malloc_mad3(target, s->img_x, s->img_y, 0);
05597    if (!out) return stbi__errpuc("outofmem", "Out of memory");
05598    if (info.bpp < 16) {
05599       int z=0;
05600       if (psize == 0 || psize > 256) { STBI_FREE(out); return stbi__errpuc("invalid", "Corrupt BMP");
     }
05601       for (i=0; i < psize; ++i) {
05602          pal[i][2] = stbi__get8(s);
05603          pal[i][1] = stbi__get8(s);
05604          pal[i][0] = stbi__get8(s);
05605          if (info.hsz != 12) stbi__get8(s);
05606          pal[i][3] = 255;
05607       }
05608       stbi__skip(s, info.offset - info.extra_read - info.hsz - psize * (info.hsz == 12 ? 3 : 4));
05609       if (info.bpp == 1) width = (s->img_x + 7) >> 3;
05610       else if (info.bpp == 4) width = (s->img_x + 1) >> 1;
05611       else if (info.bpp == 8) width = s->img_x;
05612       else { STBI_FREE(out); return stbi__errpuc("bad bpp", "Corrupt BMP"); }
05613       pad = (-width)&3;
05614       if (info.bpp == 1) {
```

```
05615          for (j=0; j < (int) s->img_y; ++j) {
05616             int bit_offset = 7, v = stbi__get8(s);
05617             for (i=0; i < (int) s->img_x; ++i) {
05618                int color = (v>>bit_offset)&0x1;
05619                out[z++] = pal[color][0];
05620                out[z++] = pal[color][1];
05621                out[z++] = pal[color][2];
05622                if (target == 4) out[z++] = 255;
05623                if (i+1 == (int) s->img_x) break;
05624                if((--bit_offset) < 0) {
05625                   bit_offset = 7;
05626                   v = stbi__get8(s);
05627                }
05628             }
05629             stbi__skip(s, pad);
05630          }
05631       } else {
05632          for (j=0; j < (int) s->img_y; ++j) {
05633             for (i=0; i < (int) s->img_x; i += 2) {
05634                int v=stbi__get8(s),v2=0;
05635                if (info.bpp == 4) {
05636                   v2 = v & 15;
05637                   v >>= 4;
05638                }
05639                out[z++] = pal[v][0];
05640                out[z++] = pal[v][1];
05641                out[z++] = pal[v][2];
05642                if (target == 4) out[z++] = 255;
05643                if (i+1 == (int) s->img_x) break;
05644                v = (info.bpp == 8) ? stbi__get8(s) : v2;
05645                out[z++] = pal[v][0];
05646                out[z++] = pal[v][1];
05647                out[z++] = pal[v][2];
05648                if (target == 4) out[z++] = 255;
05649             }
05650             stbi__skip(s, pad);
05651          }
05652       }
05653    } else {
05654       int rshift=0,gshift=0,bshift=0,ashift=0,rcount=0,gcount=0,bcount=0,acount=0;
05655       int z = 0;
05656       int easy=0;
05657       stbi__skip(s, info.offset - info.extra_read - info.hsz);
05658       if (info.bpp == 24) width = 3 * s->img_x;
05659       else if (info.bpp == 16) width = 2*s->img_x;
05660       else /* bpp = 32 and pad = 0 */ width=0;
05661       pad = (-width) & 3;
05662       if (info.bpp == 24) {
05663          easy = 1;
05664       } else if (info.bpp == 32) {
05665          if (mb == 0xff && mg == 0xff00 && mr == 0x00ff0000 && ma == 0xff000000)
05666             easy = 2;
05667       }
05668       if (!easy) {
05669          if (!mr || !mg || !mb) { STBI_FREE(out); return stbi__errpuc("bad masks", "Corrupt BMP"); }
05670          // right shift amt to put high bit in position #7
05671          rshift = stbi__high_bit(mr)-7; rcount = stbi__bitcount(mr);
05672          gshift = stbi__high_bit(mg)-7; gcount = stbi__bitcount(mg);
05673          bshift = stbi__high_bit(mb)-7; bcount = stbi__bitcount(mb);
05674          ashift = stbi__high_bit(ma)-7; acount = stbi__bitcount(ma);
05675          if (rcount > 8 || gcount > 8 || bcount > 8 || acount > 8) { STBI_FREE(out); return
     stbi__errpuc("bad masks", "Corrupt BMP"); }
05676       }
05677       for (j=0; j < (int) s->img_y; ++j) {
05678          if (easy) {
05679             for (i=0; i < (int) s->img_x; ++i) {
05680                unsigned char a;
05681                out[z+2] = stbi__get8(s);
05682                out[z+1] = stbi__get8(s);
05683                out[z+0] = stbi__get8(s);
05684                z += 3;
05685                a = (easy == 2 ? stbi__get8(s) : 255);
05686                all_a |= a;
05687                if (target == 4) out[z++] = a;
05688             }
05689          } else {
05690             int bpp = info.bpp;
05691             for (i=0; i < (int) s->img_x; ++i) {
05692                stbi__uint32 v = (bpp == 16 ? (stbi__uint32) stbi__get16le(s) : stbi__get32le(s));
05693                unsigned int a;
05694                out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mr, rshift, rcount));
05695                out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mg, gshift, gcount));
05696                out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mb, bshift, bcount));
05697                a = (ma ? stbi__shiftsigned(v & ma, ashift, acount) : 255);
05698                all_a |= a;
05699                if (target == 4) out[z++] = STBI__BYTECAST(a);
05700             }
```

```
05701          }
05702          stbi__skip(s, pad);
05703       }
05704    }
05705
05706    // if alpha channel is all 0s, replace with all 255s
05707    if (target == 4 && all_a == 0)
05708       for (i=4*s->img_x*s->img_y-1; i >= 0; i -= 4)
05709          out[i] = 255;
05710
05711    if (flip_vertically) {
05712       stbi_uc t;
05713       for (j=0; j < (int) s->img_y>>1; ++j) {
05714          stbi_uc *p1 = out +      j     *s->img_x*target;
05715          stbi_uc *p2 = out + (s->img_y-1-j)*s->img_x*target;
05716          for (i=0; i < (int) s->img_x*target; ++i) {
05717             t = p1[i]; p1[i] = p2[i]; p2[i] = t;
05718          }
05719       }
05720    }
05721
05722    if (req_comp && req_comp != target) {
05723       out = stbi__convert_format(out, target, req_comp, s->img_x, s->img_y);
05724       if (out == NULL) return out; // stbi__convert_format frees input on failure
05725    }
05726
05727    *x = s->img_x;
05728    *y = s->img_y;
05729    if (comp) *comp = s->img_n;
05730    return out;
05731 }
05732 #endif
05733
05734 // Targa Truevision - TGA
05735 // by Jonathan Dummer
05736 #ifndef STBI_NO_TGA
05737 // returns STBI_rgb or whatever, 0 on error
05738 static int stbi__tga_get_comp(int bits_per_pixel, int is_grey, int* is_rgb16)
05739 {
05740    // only RGB or RGBA (incl. 16bit) or grey allowed
05741    if (is_rgb16) *is_rgb16 = 0;
05742    switch(bits_per_pixel) {
05743       case 8:  return STBI_grey;
05744       case 16: if(is_grey) return STBI_grey_alpha;
05745                // fallthrough
05746       case 15: if(is_rgb16) *is_rgb16 = 1;
05747                return STBI_rgb;
05748       case 24: // fallthrough
05749       case 32: return bits_per_pixel/8;
05750       default: return 0;
05751    }
05752 }
05753
05754 static int stbi__tga_info(stbi__context *s, int *x, int *y, int *comp)
05755 {
05756     int tga_w, tga_h, tga_comp, tga_image_type, tga_bits_per_pixel, tga_colormap_bpp;
05757     int sz, tga_colormap_type;
05758     stbi__get8(s);                   // discard Offset
05759     tga_colormap_type = stbi__get8(s); // colormap type
05760     if( tga_colormap_type > 1 ) {
05761         stbi__rewind(s);
05762         return 0;      // only RGB or indexed allowed
05763     }
05764     tga_image_type = stbi__get8(s); // image type
05765     if ( tga_colormap_type == 1 ) { // colormapped (paletted) image
05766         if (tga_image_type != 1 && tga_image_type != 9) {
05767             stbi__rewind(s);
05768             return 0;
05769         }
05770         stbi__skip(s,4);       // skip index of first colormap entry and number of entries
05771         sz = stbi__get8(s);    //   check bits per palette color entry
05772         if ( (sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32) ) {
05773             stbi__rewind(s);
05774             return 0;
05775         }
05776         stbi__skip(s,4);       // skip image x and y origin
05777         tga_colormap_bpp = sz;
05778     } else { // "normal" image w/o colormap - only RGB or grey allowed, +/- RLE
05779         if ( (tga_image_type != 2) && (tga_image_type != 3) && (tga_image_type != 10) &&
     (tga_image_type != 11) ) {
05780             stbi__rewind(s);
05781             return 0; // only RGB or grey allowed, +/- RLE
05782         }
05783         stbi__skip(s,9); // skip colormap specification and image x/y origin
05784         tga_colormap_bpp = 0;
05785     }
05786     tga_w = stbi__get16le(s);
```

```
05787     if( tga_w < 1 ) {
05788         stbi__rewind(s);
05789         return 0;   // test width
05790     }
05791     tga_h = stbi__get16le(s);
05792     if( tga_h < 1 ) {
05793         stbi__rewind(s);
05794         return 0;   // test height
05795     }
05796     tga_bits_per_pixel = stbi__get8(s); // bits per pixel
05797     stbi__get8(s); // ignore alpha bits
05798     if (tga_colormap_bpp != 0) {
05799         if((tga_bits_per_pixel != 8) && (tga_bits_per_pixel != 16)) {
05800             // when using a colormap, tga_bits_per_pixel is the size of the indexes
05801             // I don't think anything but 8 or 16bit indexes makes sense
05802             stbi__rewind(s);
05803             return 0;
05804         }
05805         tga_comp = stbi__tga_get_comp(tga_colormap_bpp, 0, NULL);
05806     } else {
05807         tga_comp = stbi__tga_get_comp(tga_bits_per_pixel, (tga_image_type == 3) || (tga_image_type ==
      11), NULL);
05808     }
05809     if(!tga_comp) {
05810       stbi__rewind(s);
05811       return 0;
05812     }
05813     if (x) *x = tga_w;
05814     if (y) *y = tga_h;
05815     if (comp) *comp = tga_comp;
05816     return 1;                   // seems to have passed everything
05817 }
05818
05819 static int stbi__tga_test(stbi__context *s)
05820 {
05821    int res = 0;
05822    int sz, tga_color_type;
05823    stbi__get8(s);      //   discard Offset
05824    tga_color_type = stbi__get8(s);   //   color type
05825    if ( tga_color_type > 1 ) goto errorEnd;   //   only RGB or indexed allowed
05826    sz = stbi__get8(s);   //   image type
05827    if ( tga_color_type == 1 ) { // colormapped (paletted) image
05828       if (sz != 1 && sz != 9) goto errorEnd; // colortype 1 demands image type 1 or 9
05829       stbi__skip(s,4);       // skip index of first colormap entry and number of entries
05830       sz = stbi__get8(s);    //   check bits per palette color entry
05831       if ( (sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32) ) goto errorEnd;
05832       stbi__skip(s,4);       // skip image x and y origin
05833    } else { // "normal" image w/o colormap
05834       if ( (sz != 2) && (sz != 3) && (sz != 10) && (sz != 11) ) goto errorEnd; // only RGB or grey
      allowed, +/- RLE
05835       stbi__skip(s,9); // skip colormap specification and image x/y origin
05836    }
05837    if ( stbi__get16le(s) < 1 ) goto errorEnd;      //   test width
05838    if ( stbi__get16le(s) < 1 ) goto errorEnd;      //   test height
05839    sz = stbi__get8(s);   //   bits per pixel
05840    if ( (tga_color_type == 1) && (sz != 8) && (sz != 16) ) goto errorEnd; // for colormapped images,
      bpp is size of an index
05841    if ( (sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32) ) goto errorEnd;
05842
05843    res = 1; // if we got this far, everything's good and we can return 1 instead of 0
05844
05845 errorEnd:
05846    stbi__rewind(s);
05847    return res;
05848 }
05849
05850 // read 16bit value and convert to 24bit RGB
05851 static void stbi__tga_read_rgb16(stbi__context *s, stbi_uc* out)
05852 {
05853    stbi__uint16 px = (stbi__uint16)stbi__get16le(s);
05854    stbi__uint16 fiveBitMask = 31;
05855    // we have 3 channels with 5bits each
05856    int r = (px >> 10) & fiveBitMask;
05857    int g = (px >> 5) & fiveBitMask;
05858    int b = px & fiveBitMask;
05859    // Note that this saves the data in RGB(A) order, so it doesn't need to be swapped later
05860    out[0] = (stbi_uc)((r * 255)/31);
05861    out[1] = (stbi_uc)((g * 255)/31);
05862    out[2] = (stbi_uc)((b * 255)/31);
05863
05864    // some people claim that the most significant bit might be used for alpha
05865    // (possibly if an alpha-bit is set in the "image descriptor byte")
05866    // but that only made 16bit test images completely translucent..
05867    // so let's treat all 15 and 16bit TGAs as RGB with no alpha.
05868 }
05869
05870 static void *stbi__tga_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
```

```
      stbi__result_info *ri)
05871 {
05872    //   read in the TGA header stuff
05873    int tga_offset = stbi__get8(s);
05874    int tga_indexed = stbi__get8(s);
05875    int tga_image_type = stbi__get8(s);
05876    int tga_is_RLE = 0;
05877    int tga_palette_start = stbi__get16le(s);
05878    int tga_palette_len = stbi__get16le(s);
05879    int tga_palette_bits = stbi__get8(s);
05880    int tga_x_origin = stbi__get16le(s);
05881    int tga_y_origin = stbi__get16le(s);
05882    int tga_width = stbi__get16le(s);
05883    int tga_height = stbi__get16le(s);
05884    int tga_bits_per_pixel = stbi__get8(s);
05885    int tga_comp, tga_rgb16=0;
05886    int tga_inverted = stbi__get8(s);
05887    // int tga_alpha_bits = tga_inverted & 15; // the 4 lowest bits - unused (useless?)
05888    //   image data
05889    unsigned char *tga_data;
05890    unsigned char *tga_palette = NULL;
05891    int i, j;
05892    unsigned char raw_data[4] = {0};
05893    int RLE_count = 0;
05894    int RLE_repeating = 0;
05895    int read_next_pixel = 1;
05896    STBI_NOTUSED(ri);
05897    STBI_NOTUSED(tga_x_origin); // @TODO
05898    STBI_NOTUSED(tga_y_origin); // @TODO
05899
05900    if (tga_height > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image
      (corrupt?)");
05901    if (tga_width > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image
      (corrupt?)");
05902
05903    //   do a tiny bit of precessing
05904    if ( tga_image_type >= 8 )
05905    {
05906       tga_image_type -= 8;
05907       tga_is_RLE = 1;
05908    }
05909    tga_inverted = 1 - ((tga_inverted >> 5) & 1);
05910
05911    //   If I'm paletted, then I'll use the number of bits from the palette
05912    if ( tga_indexed ) tga_comp = stbi__tga_get_comp(tga_palette_bits, 0, &tga_rgb16);
05913    else tga_comp = stbi__tga_get_comp(tga_bits_per_pixel, (tga_image_type == 3), &tga_rgb16);
05914
05915    if(!tga_comp) // shouldn't really happen, stbi__tga_test() should have ensured basic consistency
05916       return stbi__errpuc("bad format", "Can't find out TGA pixelformat");
05917
05918    //   tga info
05919    *x = tga_width;
05920    *y = tga_height;
05921    if (comp) *comp = tga_comp;
05922
05923    if (!stbi__mad3sizes_valid(tga_width, tga_height, tga_comp, 0))
05924       return stbi__errpuc("too large", "Corrupt TGA");
05925
05926    tga_data = (unsigned char*)stbi__malloc_mad3(tga_width, tga_height, tga_comp, 0);
05927    if (!tga_data) return stbi__errpuc("outofmem", "Out of memory");
05928
05929    // skip to the data's starting position (offset usually = 0)
05930    stbi__skip(s, tga_offset );
05931
05932    if ( !tga_indexed && !tga_is_RLE && !tga_rgb16 ) {
05933       for (i=0; i < tga_height; ++i) {
05934          int row = tga_inverted ? tga_height -i - 1 : i;
05935          stbi_uc *tga_row = tga_data + row*tga_width*tga_comp;
05936          stbi__getn(s, tga_row, tga_width * tga_comp);
05937       }
05938    } else  {
05939       //   do I need to load a palette?
05940       if ( tga_indexed)
05941       {
05942          if (tga_palette_len == 0) {  /* you have to have at least one entry! */
05943             STBI_FREE(tga_data);
05944             return stbi__errpuc("bad palette", "Corrupt TGA");
05945          }
05946
05947          //   any data to skip? (offset usually = 0)
05948          stbi__skip(s, tga_palette_start );
05949          //   load the palette
05950          tga_palette = (unsigned char*)stbi__malloc_mad2(tga_palette_len, tga_comp, 0);
05951          if (!tga_palette) {
05952             STBI_FREE(tga_data);
05953             return stbi__errpuc("outofmem", "Out of memory");
05954          }
```

```
05955          if (tga_rgb16) {
05956             stbi_uc *pal_entry = tga_palette;
05957             STBI_ASSERT(tga_comp == STBI_rgb);
05958             for (i=0; i < tga_palette_len; ++i) {
05959                stbi__tga_read_rgb16(s, pal_entry);
05960                pal_entry += tga_comp;
05961             }
05962          } else if (!stbi__getn(s, tga_palette, tga_palette_len * tga_comp)) {
05963                STBI_FREE(tga_data);
05964                STBI_FREE(tga_palette);
05965                return stbi__errpuc("bad palette", "Corrupt TGA");
05966          }
05967       }
05968       //   load the data
05969       for (i=0; i < tga_width * tga_height; ++i)
05970       {
05971          //   if I'm in RLE mode, do I need to get a RLE stbi__pngchunk?
05972          if ( tga_is_RLE )
05973          {
05974             if ( RLE_count == 0 )
05975             {
05976                //   yep, get the next byte as a RLE command
05977                int RLE_cmd = stbi__get8(s);
05978                RLE_count = 1 + (RLE_cmd & 127);
05979                RLE_repeating = RLE_cmd >> 7;
05980                read_next_pixel = 1;
05981             } else if ( !RLE_repeating )
05982             {
05983                read_next_pixel = 1;
05984             }
05985          } else
05986          {
05987             read_next_pixel = 1;
05988          }
05989          //   OK, if I need to read a pixel, do it now
05990          if ( read_next_pixel )
05991          {
05992             //   load however much data we did have
05993             if ( tga_indexed )
05994             {
05995                // read in index, then perform the lookup
05996                int pal_idx = (tga_bits_per_pixel == 8) ? stbi__get8(s) : stbi__get16le(s);
05997                if ( pal_idx >= tga_palette_len ) {
05998                   // invalid index
05999                   pal_idx = 0;
06000                }
06001                pal_idx *= tga_comp;
06002                for (j = 0; j < tga_comp; ++j) {
06003                   raw_data[j] = tga_palette[pal_idx+j];
06004                }
06005             } else if(tga_rgb16) {
06006                STBI_ASSERT(tga_comp == STBI_rgb);
06007                stbi__tga_read_rgb16(s, raw_data);
06008             } else {
06009                //   read in the data raw
06010                for (j = 0; j < tga_comp; ++j) {
06011                   raw_data[j] = stbi__get8(s);
06012                }
06013             }
06014             //   clear the reading flag for the next pixel
06015             read_next_pixel = 0;
06016          } // end of reading a pixel
06017
06018          // copy data
06019          for (j = 0; j < tga_comp; ++j)
06020            tga_data[i*tga_comp+j] = raw_data[j];
06021
06022          //   in case we're in RLE mode, keep counting down
06023          --RLE_count;
06024       }
06025       //   do I need to invert the image?
06026       if ( tga_inverted )
06027       {
06028          for (j = 0; j*2 < tga_height; ++j)
06029          {
06030             int index1 = j * tga_width * tga_comp;
06031             int index2 = (tga_height - 1 - j) * tga_width * tga_comp;
06032             for (i = tga_width * tga_comp; i > 0; --i)
06033             {
06034                unsigned char temp = tga_data[index1];
06035                tga_data[index1] = tga_data[index2];
06036                tga_data[index2] = temp;
06037                ++index1;
06038                ++index2;
06039             }
06040          }
06041       }
```

```
06042       //   clear my palette, if I had one
06043       if ( tga_palette != NULL )
06044       {
06045          STBI_FREE( tga_palette );
06046       }
06047    }
06048
06049    // swap RGB - if the source data was RGB16, it already is in the right order
06050    if (tga_comp >= 3 && !tga_rgb16)
06051    {
06052       unsigned char* tga_pixel = tga_data;
06053       for (i=0; i < tga_width * tga_height; ++i)
06054       {
06055          unsigned char temp = tga_pixel[0];
06056          tga_pixel[0] = tga_pixel[2];
06057          tga_pixel[2] = temp;
06058          tga_pixel += tga_comp;
06059       }
06060    }
06061
06062    // convert to target component count
06063    if (req_comp && req_comp != tga_comp)
06064       tga_data = stbi__convert_format(tga_data, tga_comp, req_comp, tga_width, tga_height);
06065
06066    //   the things I do to get rid of an error message, and yet keep
06067    //   Microsoft's C compilers happy... [8^(
06068    tga_palette_start = tga_palette_len = tga_palette_bits =
06069          tga_x_origin = tga_y_origin = 0;
06070    STBI_NOTUSED(tga_palette_start);
06071    //   OK, done
06072    return tga_data;
06073 }
06074 #endif
06075
06076 // *************************************************************************************************
06077 // Photoshop PSD loader -- PD by Thatcher Ulrich, integration by Nicolas Schulz, tweaked by STB
06078
06079 #ifndef STBI_NO_PSD
06080 static int stbi__psd_test(stbi__context *s)
06081 {
06082    int r = (stbi__get32be(s) == 0x38425053);
06083    stbi__rewind(s);
06084    return r;
06085 }
06086
06087 static int stbi__psd_decode_rle(stbi__context *s, stbi_uc *p, int pixelCount)
06088 {
06089    int count, nleft, len;
06090
06091    count = 0;
06092    while ((nleft = pixelCount - count) > 0) {
06093       len = stbi__get8(s);
06094       if (len == 128) {
06095          // No-op.
06096       } else if (len < 128) {
06097          // Copy next len+1 bytes literally.
06098          len++;
06099          if (len > nleft) return 0; // corrupt data
06100          count += len;
06101          while (len) {
06102             *p = stbi__get8(s);
06103             p += 4;
06104             len--;
06105          }
06106       } else if (len > 128) {
06107          stbi_uc   val;
06108          // Next -len+1 bytes in the dest are replicated from next source byte.
06109          // (Interpret len as a negative 8-bit int.)
06110          len = 257 - len;
06111          if (len > nleft) return 0; // corrupt data
06112          val = stbi__get8(s);
06113          count += len;
06114          while (len) {
06115             *p = val;
06116             p += 4;
06117             len--;
06118          }
06119       }
06120    }
06121
06122    return 1;
06123 }
06124
06125 static void *stbi__psd_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
       stbi__result_info *ri, int bpc)
06126 {
06127    int pixelCount;
```

```
06128    int channelCount, compression;
06129    int channel, i;
06130    int bitdepth;
06131    int w,h;
06132    stbi_uc *out;
06133    STBI_NOTUSED(ri);
06134
06135    // Check identifier
06136    if (stbi__get32be(s) != 0x38425053)   // "8BPS"
06137       return stbi__errpuc("not PSD", "Corrupt PSD image");
06138
06139    // Check file type version.
06140    if (stbi__get16be(s) != 1)
06141       return stbi__errpuc("wrong version", "Unsupported version of PSD image");
06142
06143    // Skip 6 reserved bytes.
06144    stbi__skip(s, 6 );
06145
06146    // Read the number of channels (R, G, B, A, etc).
06147    channelCount = stbi__get16be(s);
06148    if (channelCount < 0 || channelCount > 16)
06149       return stbi__errpuc("wrong channel count", "Unsupported number of channels in PSD image");
06150
06151    // Read the rows and columns of the image.
06152    h = stbi__get32be(s);
06153    w = stbi__get32be(s);
06154
06155    if (h > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
06156    if (w > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
06157
06158    // Make sure the depth is 8 bits.
06159    bitdepth = stbi__get16be(s);
06160    if (bitdepth != 8 && bitdepth != 16)
06161       return stbi__errpuc("unsupported bit depth", "PSD bit depth is not 8 or 16 bit");
06162
06163    // Make sure the color mode is RGB.
06164    // Valid options are:
06165    //   0: Bitmap
06166    //   1: Grayscale
06167    //   2: Indexed color
06168    //   3: RGB color
06169    //   4: CMYK color
06170    //   7: Multichannel
06171    //   8: Duotone
06172    //   9: Lab color
06173    if (stbi__get16be(s) != 3)
06174       return stbi__errpuc("wrong color format", "PSD is not in RGB color format");
06175
06176    // Skip the Mode Data.  (It's the palette for indexed color; other info for other modes.)
06177    stbi__skip(s,stbi__get32be(s) );
06178
06179    // Skip the image resources.  (resolution, pen tool paths, etc)
06180    stbi__skip(s, stbi__get32be(s) );
06181
06182    // Skip the reserved data.
06183    stbi__skip(s, stbi__get32be(s) );
06184
06185    // Find out if the data is compressed.
06186    // Known values:
06187    //   0: no compression
06188    //   1: RLE compressed
06189    compression = stbi__get16be(s);
06190    if (compression > 1)
06191       return stbi__errpuc("bad compression", "PSD has an unknown compression format");
06192
06193    // Check size
06194    if (!stbi__mad3sizes_valid(4, w, h, 0))
06195       return stbi__errpuc("too large", "Corrupt PSD");
06196
06197    // Create the destination image.
06198
06199    if (!compression && bitdepth == 16 && bpc == 16) {
06200       out = (stbi_uc *) stbi__malloc_mad3(8, w, h, 0);
06201       ri->bits_per_channel = 16;
06202    } else
06203       out = (stbi_uc *) stbi__malloc(4 * w*h);
06204
06205    if (!out) return stbi__errpuc("outofmem", "Out of memory");
06206    pixelCount = w*h;
06207
06208    // Initialize the data to zero.
06209    //memset( out, 0, pixelCount * 4 );
06210
06211    // Finally, the image data.
06212    if (compression) {
06213       // RLE as used by .PSD and .TIFF
06214       // Loop until you get the number of unpacked bytes you are expecting:
```

```
06215       //     Read the next source byte into n.
06216       //     If n is between 0 and 127 inclusive, copy the next n+1 bytes literally.
06217       //     Else if n is between -127 and -1 inclusive, copy the next byte -n+1 times.
06218       //     Else if n is 128, noop.
06219       // Endloop
06220
06221       // The RLE-compressed data is preceded by a 2-byte data count for each row in the data,
06222       // which we're going to just skip.
06223       stbi__skip(s, h * channelCount * 2 );
06224
06225       // Read the RLE data by channel.
06226       for (channel = 0; channel < 4; channel++) {
06227          stbi_uc *p;
06228
06229          p = out+channel;
06230          if (channel >= channelCount) {
06231             // Fill this channel with default data.
06232             for (i = 0; i < pixelCount; i++, p += 4)
06233                *p = (channel == 3 ? 255 : 0);
06234          } else {
06235             // Read the RLE data.
06236             if (!stbi__psd_decode_rle(s, p, pixelCount)) {
06237                STBI_FREE(out);
06238                return stbi__errpuc("corrupt", "bad RLE data");
06239             }
06240          }
06241       }
06242
06243    } else {
06244       // We're at the raw image data.  It's each channel in order (Red, Green, Blue, Alpha, ...)
06245       // where each channel consists of an 8-bit (or 16-bit) value for each pixel in the image.
06246
06247       // Read the data by channel.
06248       for (channel = 0; channel < 4; channel++) {
06249          if (channel >= channelCount) {
06250             // Fill this channel with default data.
06251             if (bitdepth == 16 && bpc == 16) {
06252                stbi__uint16 *q = ((stbi__uint16 *) out) + channel;
06253                stbi__uint16 val = channel == 3 ? 65535 : 0;
06254                for (i = 0; i < pixelCount; i++, q += 4)
06255                   *q = val;
06256             } else {
06257                stbi_uc *p = out+channel;
06258                stbi_uc val = channel == 3 ? 255 : 0;
06259                for (i = 0; i < pixelCount; i++, p += 4)
06260                   *p = val;
06261             }
06262          } else {
06263             if (ri->bits_per_channel == 16) {    // output bpc
06264                stbi__uint16 *q = ((stbi__uint16 *) out) + channel;
06265                for (i = 0; i < pixelCount; i++, q += 4)
06266                   *q = (stbi__uint16) stbi__get16be(s);
06267             } else {
06268                stbi_uc *p = out+channel;
06269                if (bitdepth == 16) {  // input bpc
06270                   for (i = 0; i < pixelCount; i++, p += 4)
06271                      *p = (stbi_uc) (stbi__get16be(s) » 8);
06272                } else {
06273                   for (i = 0; i < pixelCount; i++, p += 4)
06274                      *p = stbi__get8(s);
06275                }
06276             }
06277          }
06278       }
06279    }
06280
06281    // remove weird white matte from PSD
06282    if (channelCount >= 4) {
06283       if (ri->bits_per_channel == 16) {
06284          for (i=0; i < w*h; ++i) {
06285             stbi__uint16 *pixel = (stbi__uint16 *) out + 4*i;
06286             if (pixel[3] != 0 && pixel[3] != 65535) {
06287                float a = pixel[3] / 65535.0f;
06288                float ra = 1.0f / a;
06289                float inv_a = 65535.0f * (1 - ra);
06290                pixel[0] = (stbi__uint16) (pixel[0]*ra + inv_a);
06291                pixel[1] = (stbi__uint16) (pixel[1]*ra + inv_a);
06292                pixel[2] = (stbi__uint16) (pixel[2]*ra + inv_a);
06293             }
06294          }
06295       } else {
06296          for (i=0; i < w*h; ++i) {
06297             unsigned char *pixel = out + 4*i;
06298             if (pixel[3] != 0 && pixel[3] != 255) {
06299                float a = pixel[3] / 255.0f;
06300                float ra = 1.0f / a;
06301                float inv_a = 255.0f * (1 - ra);
```

```
06302                pixel[0] = (unsigned char) (pixel[0]*ra + inv_a);
06303                pixel[1] = (unsigned char) (pixel[1]*ra + inv_a);
06304                pixel[2] = (unsigned char) (pixel[2]*ra + inv_a);
06305             }
06306          }
06307       }
06308    }
06309
06310    // convert to desired output format
06311    if (req_comp && req_comp != 4) {
06312       if (ri->bits_per_channel == 16)
06313          out = (stbi_uc *) stbi__convert_format16((stbi__uint16 *) out, 4, req_comp, w, h);
06314       else
06315          out = stbi__convert_format(out, 4, req_comp, w, h);
06316       if (out == NULL) return out; // stbi__convert_format frees input on failure
06317    }
06318
06319    if (comp) *comp = 4;
06320    *y = h;
06321    *x = w;
06322
06323    return out;
06324 }
06325 #endif
06326
06327 // ********************************************************************************************************
06328 // Softimage PIC loader
06329 // by Tom Seddon
06330 //
06331 // See http://softimage.wiki.softimage.com/index.php/INFO:_PIC_file_format
06332 // See http://ozviz.wasp.uwa.edu.au/~pbourke/dataformats/softimagepic/
06333
06334 #ifndef STBI_NO_PIC
06335 static int stbi__pic_is4(stbi__context *s,const char *str)
06336 {
06337    int i;
06338    for (i=0; i<4; ++i)
06339       if (stbi__get8(s) != (stbi_uc)str[i])
06340          return 0;
06341
06342    return 1;
06343 }
06344
06345 static int stbi__pic_test_core(stbi__context *s)
06346 {
06347    int i;
06348
06349    if (!stbi__pic_is4(s,"\x53\x80\xF6\x34"))
06350       return 0;
06351
06352    for(i=0;i<84;++i)
06353       stbi__get8(s);
06354
06355    if (!stbi__pic_is4(s,"PICT"))
06356       return 0;
06357
06358    return 1;
06359 }
06360
06361 typedef struct
06362 {
06363    stbi_uc size,type,channel;
06364 } stbi__pic_packet;
06365
06366 static stbi_uc *stbi__readval(stbi__context *s, int channel, stbi_uc *dest)
06367 {
06368    int mask=0x80, i;
06369
06370    for (i=0; i<4; ++i, mask>>=1) {
06371       if (channel & mask) {
06372          if (stbi__at_eof(s)) return stbi__errpuc("bad file","PIC file too short");
06373          dest[i]=stbi__get8(s);
06374       }
06375    }
06376
06377    return dest;
06378 }
06379
06380 static void stbi__copyval(int channel,stbi_uc *dest,const stbi_uc *src)
06381 {
06382    int mask=0x80,i;
06383
06384    for (i=0;i<4; ++i, mask>>=1)
06385       if (channel&mask)
06386          dest[i]=src[i];
06387 }
06388
```

```
06389 static stbi_uc *stbi__pic_load_core(stbi__context *s,int width,int height,int *comp, stbi_uc *result)
06390 {
06391    int act_comp=0,num_packets=0,y,chained;
06392    stbi__pic_packet packets[10];
06393
06394    // this will (should...) cater for even some bizarre stuff like having data
06395     // for the same channel in multiple packets.
06396    do {
06397       stbi__pic_packet *packet;
06398
06399       if (num_packets==sizeof(packets)/sizeof(packets[0]))
06400          return stbi__errpuc("bad format","too many packets");
06401
06402       packet = &packets[num_packets++];
06403
06404       chained = stbi__get8(s);
06405       packet->size    = stbi__get8(s);
06406       packet->type    = stbi__get8(s);
06407       packet->channel = stbi__get8(s);
06408
06409       act_comp |= packet->channel;
06410
06411       if (stbi__at_eof(s))          return stbi__errpuc("bad file","file too short (reading
      packets)");
06412       if (packet->size != 8)  return stbi__errpuc("bad format","packet isn't 8bpp");
06413    } while (chained);
06414
06415    *comp = (act_comp & 0x10 ? 4 : 3); // has alpha channel?
06416
06417    for(y=0; y<height; ++y) {
06418       int packet_idx;
06419
06420       for(packet_idx=0; packet_idx < num_packets; ++packet_idx) {
06421          stbi__pic_packet *packet = &packets[packet_idx];
06422          stbi_uc *dest = result+y*width*4;
06423
06424          switch (packet->type) {
06425             default:
06426                return stbi__errpuc("bad format","packet has bad compression type");
06427
06428             case 0: {//uncompressed
06429                int x;
06430
06431                for(x=0;x<width;++x, dest+=4)
06432                   if (!stbi__readval(s,packet->channel,dest))
06433                      return 0;
06434                break;
06435             }
06436
06437             case 1://Pure RLE
06438                {
06439                   int left=width, i;
06440
06441                   while (left>0) {
06442                      stbi_uc count,value[4];
06443
06444                      count=stbi__get8(s);
06445                      if (stbi__at_eof(s))   return stbi__errpuc("bad file","file too short (pure read
      count)");
06446
06447                      if (count > left)
06448                         count = (stbi_uc) left;
06449
06450                      if (!stbi__readval(s,packet->channel,value))  return 0;
06451
06452                      for(i=0; i<count; ++i,dest+=4)
06453                         stbi__copyval(packet->channel,dest,value);
06454                      left -= count;
06455                   }
06456                }
06457                break;
06458
06459             case 2: {//Mixed RLE
06460                int left=width;
06461                while (left>0) {
06462                   int count = stbi__get8(s), i;
06463                   if (stbi__at_eof(s))  return stbi__errpuc("bad file","file too short (mixed read
      count)");
06464
06465                   if (count >= 128) { // Repeated
06466                      stbi_uc value[4];
06467
06468                      if (count==128)
06469                         count = stbi__get16be(s);
06470                      else
06471                         count -= 127;
06472                      if (count > left)
```

```
06473                         return stbi__errpuc("bad file","scanline overrun");
06474
06475                      if (!stbi__readval(s,packet->channel,value))
06476                         return 0;
06477
06478                      for(i=0;i<count;++i, dest += 4)
06479                         stbi__copyval(packet->channel,dest,value);
06480                   } else { // Raw
06481                      ++count;
06482                      if (count>left) return stbi__errpuc("bad file","scanline overrun");
06483
06484                      for(i=0;i<count;++i, dest+=4)
06485                         if (!stbi__readval(s,packet->channel,dest))
06486                            return 0;
06487                   }
06488                   left-=count;
06489                }
06490                break;
06491             }
06492          }
06493       }
06494    }
06495
06496    return result;
06497 }
06498
06499 static void *stbi__pic_load(stbi__context *s,int *px,int *py,int *comp,int req_comp, stbi__result_info
      *ri)
06500 {
06501    stbi_uc *result;
06502    int i, x,y, internal_comp;
06503    STBI_NOTUSED(ri);
06504
06505    if (!comp) comp = &internal_comp;
06506
06507    for (i=0; i<92; ++i)
06508       stbi__get8(s);
06509
06510    x = stbi__get16be(s);
06511    y = stbi__get16be(s);
06512
06513    if (y > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
06514    if (x > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
06515
06516    if (stbi__at_eof(s))  return stbi__errpuc("bad file","file too short (pic header)");
06517    if (!stbi__mad3sizes_valid(x, y, 4, 0)) return stbi__errpuc("too large", "PIC image too large to
      decode");
06518
06519    stbi__get32be(s); //skip 'ratio'
06520    stbi__get16be(s); //skip 'fields'
06521    stbi__get16be(s); //skip 'pad'
06522
06523    // intermediate buffer is RGBA
06524    result = (stbi_uc *) stbi__malloc_mad3(x, y, 4, 0);
06525    if (!result) return stbi__errpuc("outofmem", "Out of memory");
06526    memset(result, 0xff, x*y*4);
06527
06528    if (!stbi__pic_load_core(s,x,y,comp, result)) {
06529       STBI_FREE(result);
06530       result=0;
06531    }
06532    *px = x;
06533    *py = y;
06534    if (req_comp == 0) req_comp = *comp;
06535    result=stbi__convert_format(result,4,req_comp,x,y);
06536
06537    return result;
06538 }
06539
06540 static int stbi__pic_test(stbi__context *s)
06541 {
06542    int r = stbi__pic_test_core(s);
06543    stbi__rewind(s);
06544    return r;
06545 }
06546 #endif
06547
06548 // ************************************************************************************************
06549 // GIF loader -- public domain by Jean-Marc Lienher -- simplified/shrunk by stb
06550
06551 #ifndef STBI_NO_GIF
06552 typedef struct
06553 {
06554    stbi__int16 prefix;
06555    stbi_uc first;
06556    stbi_uc suffix;
06557 } stbi__gif_lzw;
```

```
06558
06559 typedef struct
06560 {
06561    int w,h;
06562    stbi_uc *out;                // output buffer (always 4 components)
06563    stbi_uc *background;         // The current "background" as far as a gif is concerned
06564    stbi_uc *history;
06565    int flags, bgindex, ratio, transparent, eflags;
06566    stbi_uc  pal[256][4];
06567    stbi_uc lpal[256][4];
06568    stbi__gif_lzw codes[8192];
06569    stbi_uc *color_table;
06570    int parse, step;
06571    int lflags;
06572    int start_x, start_y;
06573    int max_x, max_y;
06574    int cur_x, cur_y;
06575    int line_size;
06576    int delay;
06577 } stbi__gif;
06578
06579 static int stbi__gif_test_raw(stbi__context *s)
06580 {
06581    int sz;
06582    if (stbi__get8(s) != 'G' || stbi__get8(s) != 'I' || stbi__get8(s) != 'F' || stbi__get8(s) != '8')
     return 0;
06583    sz = stbi__get8(s);
06584    if (sz != '9' && sz != '7') return 0;
06585    if (stbi__get8(s) != 'a') return 0;
06586    return 1;
06587 }
06588
06589 static int stbi__gif_test(stbi__context *s)
06590 {
06591    int r = stbi__gif_test_raw(s);
06592    stbi__rewind(s);
06593    return r;
06594 }
06595
06596 static void stbi__gif_parse_colortable(stbi__context *s, stbi_uc pal[256][4], int num_entries, int
     transp)
06597 {
06598    int i;
06599    for (i=0; i < num_entries; ++i) {
06600       pal[i][2] = stbi__get8(s);
06601       pal[i][1] = stbi__get8(s);
06602       pal[i][0] = stbi__get8(s);
06603       pal[i][3] = transp == i ? 0 : 255;
06604    }
06605 }
06606
06607 static int stbi__gif_header(stbi__context *s, stbi__gif *g, int *comp, int is_info)
06608 {
06609    stbi_uc version;
06610    if (stbi__get8(s) != 'G' || stbi__get8(s) != 'I' || stbi__get8(s) != 'F' || stbi__get8(s) != '8')
06611       return stbi__err("not GIF", "Corrupt GIF");
06612
06613    version = stbi__get8(s);
06614    if (version != '7' && version != '9')    return stbi__err("not GIF", "Corrupt GIF");
06615    if (stbi__get8(s) != 'a')                return stbi__err("not GIF", "Corrupt GIF");
06616
06617    stbi__g_failure_reason = "";
06618    g->w = stbi__get16le(s);
06619    g->h = stbi__get16le(s);
06620    g->flags = stbi__get8(s);
06621    g->bgindex = stbi__get8(s);
06622    g->ratio = stbi__get8(s);
06623    g->transparent = -1;
06624
06625    if (g->w > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image (corrupt?)");
06626    if (g->h > STBI_MAX_DIMENSIONS) return stbi__err("too large","Very large image (corrupt?)");
06627
06628    if (comp != 0) *comp = 4;  // can't actually tell whether it's 3 or 4 until we parse the comments
06629
06630    if (is_info) return 1;
06631
06632    if (g->flags & 0x80)
06633       stbi__gif_parse_colortable(s,g->pal, 2 « (g->flags & 7), -1);
06634
06635    return 1;
06636 }
06637
06638 static int stbi__gif_info_raw(stbi__context *s, int *x, int *y, int *comp)
06639 {
06640    stbi__gif* g = (stbi__gif*) stbi__malloc(sizeof(stbi__gif));
06641    if (!g) return stbi__err("outofmem", "Out of memory");
06642    if (!stbi__gif_header(s, g, comp, 1)) {
```

```
06643       STBI_FREE(g);
06644       stbi__rewind( s );
06645       return 0;
06646    }
06647    if (x) *x = g->w;
06648    if (y) *y = g->h;
06649    STBI_FREE(g);
06650    return 1;
06651 }
06652
06653 static void stbi__out_gif_code(stbi__gif *g, stbi__uint16 code)
06654 {
06655    stbi_uc *p, *c;
06656    int idx;
06657
06658    // recurse to decode the prefixes, since the linked-list is backwards,
06659    // and working backwards through an interleaved image would be nasty
06660    if (g->codes[code].prefix >= 0)
06661       stbi__out_gif_code(g, g->codes[code].prefix);
06662
06663    if (g->cur_y >= g->max_y) return;
06664
06665    idx = g->cur_x + g->cur_y;
06666    p = &g->out[idx];
06667    g->history[idx / 4] = 1;
06668
06669    c = &g->color_table[g->codes[code].suffix * 4];
06670    if (c[3] > 128) { // don't render transparent pixels;
06671       p[0] = c[2];
06672       p[1] = c[1];
06673       p[2] = c[0];
06674       p[3] = c[3];
06675    }
06676    g->cur_x += 4;
06677
06678    if (g->cur_x >= g->max_x) {
06679       g->cur_x = g->start_x;
06680       g->cur_y += g->step;
06681
06682       while (g->cur_y >= g->max_y && g->parse > 0) {
06683          g->step = (1 << g->parse) * g->line_size;
06684          g->cur_y = g->start_y + (g->step >> 1);
06685          --g->parse;
06686       }
06687    }
06688 }
06689
06690 static stbi_uc *stbi__process_gif_raster(stbi__context *s, stbi__gif *g)
06691 {
06692    stbi_uc lzw_cs;
06693    stbi__int32 len, init_code;
06694    stbi__uint32 first;
06695    stbi__int32 codesize, codemask, avail, oldcode, bits, valid_bits, clear;
06696    stbi__gif_lzw *p;
06697
06698    lzw_cs = stbi__get8(s);
06699    if (lzw_cs > 12) return NULL;
06700    clear = 1 << lzw_cs;
06701    first = 1;
06702    codesize = lzw_cs + 1;
06703    codemask = (1 << codesize) - 1;
06704    bits = 0;
06705    valid_bits = 0;
06706    for (init_code = 0; init_code < clear; init_code++) {
06707       g->codes[init_code].prefix = -1;
06708       g->codes[init_code].first = (stbi_uc) init_code;
06709       g->codes[init_code].suffix = (stbi_uc) init_code;
06710    }
06711
06712    // support no starting clear code
06713    avail = clear+2;
06714    oldcode = -1;
06715
06716    len = 0;
06717    for(;;) {
06718       if (valid_bits < codesize) {
06719          if (len == 0) {
06720             len = stbi__get8(s); // start new block
06721             if (len == 0)
06722                return g->out;
06723          }
06724          --len;
06725          bits |= (stbi__int32) stbi__get8(s) << valid_bits;
06726          valid_bits += 8;
06727       } else {
06728          stbi__int32 code = bits & codemask;
06729          bits >>= codesize;
```

```
06730          valid_bits -= codesize;
06731          // @OPTIMIZE: is there some way we can accelerate the non-clear path?
06732          if (code == clear) {  // clear code
06733             codesize = lzw_cs + 1;
06734             codemask = (1 « codesize) - 1;
06735             avail = clear + 2;
06736             oldcode = -1;
06737             first = 0;
06738          } else if (code == clear + 1) { // end of stream code
06739             stbi__skip(s, len);
06740             while ((len = stbi__get8(s)) > 0)
06741                stbi__skip(s,len);
06742             return g->out;
06743          } else if (code <= avail) {
06744             if (first) {
06745                return stbi__errpuc("no clear code", "Corrupt GIF");
06746             }
06747
06748             if (oldcode >= 0) {
06749                p = &g->codes[avail++];
06750                if (avail > 8192) {
06751                   return stbi__errpuc("too many codes", "Corrupt GIF");
06752                }
06753
06754                p->prefix = (stbi__int16) oldcode;
06755                p->first = g->codes[oldcode].first;
06756                p->suffix = (code == avail) ? p->first : g->codes[code].first;
06757             } else if (code == avail)
06758                return stbi__errpuc("illegal code in raster", "Corrupt GIF");
06759
06760             stbi__out_gif_code(g, (stbi__uint16) code);
06761
06762             if ((avail & codemask) == 0 && avail <= 0x0FFF) {
06763                codesize++;
06764                codemask = (1 « codesize) - 1;
06765             }
06766
06767             oldcode = code;
06768          } else {
06769             return stbi__errpuc("illegal code in raster", "Corrupt GIF");
06770          }
06771       }
06772    }
06773 }
06774
06775 // this function is designed to support animated gifs, although stb_image doesn't support it
06776 // two back is the image from two frames ago, used for a very specific disposal format
06777 static stbi_uc *stbi__gif_load_next(stbi__context *s, stbi__gif *g, int *comp, int req_comp, stbi_uc
      *two_back)
06778 {
06779    int dispose;
06780    int first_frame;
06781    int pi;
06782    int pcount;
06783    STBI_NOTUSED(req_comp);
06784
06785    // on first frame, any non-written pixels get the background colour (non-transparent)
06786    first_frame = 0;
06787    if (g->out == 0) {
06788       if (!stbi__gif_header(s, g, comp,0)) return 0; // stbi__g_failure_reason set by stbi__gif_header
06789       if (!stbi__mad3sizes_valid(4, g->w, g->h, 0))
06790          return stbi__errpuc("too large", "GIF image is too large");
06791       pcount = g->w * g->h;
06792       g->out = (stbi_uc *) stbi__malloc(4 * pcount);
06793       g->background = (stbi_uc *) stbi__malloc(4 * pcount);
06794       g->history = (stbi_uc *) stbi__malloc(pcount);
06795       if (!g->out || !g->background || !g->history)
06796          return stbi__errpuc("outofmem", "Out of memory");
06797
06798       // image is treated as "transparent" at the start - ie, nothing overwrites the current
      background;
06799       // background colour is only used for pixels that are not rendered first frame, after that
      "background"
06800       // color refers to the color that was there the previous frame.
06801       memset(g->out, 0x00, 4 * pcount);
06802       memset(g->background, 0x00, 4 * pcount); // state of the background (starts transparent)
06803       memset(g->history, 0x00, pcount);        // pixels that were affected previous frame
06804       first_frame = 1;
06805    } else {
06806       // second frame - how do we dispose of the previous one?
06807       dispose = (g->eflags & 0x1C) » 2;
06808       pcount = g->w * g->h;
06809
06810       if ((dispose == 3) && (two_back == 0)) {
06811          dispose = 2; // if I don't have an image to revert back to, default to the old background
06812       }
06813
```

```
06814       if (dispose == 3) { // use previous graphic
06815          for (pi = 0; pi < pcount; ++pi) {
06816             if (g->history[pi]) {
06817                memcpy( &g->out[pi * 4], &two_back[pi * 4], 4 );
06818             }
06819          }
06820       } else if (dispose == 2) {
06821          // restore what was changed last frame to background before that frame;
06822          for (pi = 0; pi < pcount; ++pi) {
06823             if (g->history[pi]) {
06824                memcpy( &g->out[pi * 4], &g->background[pi * 4], 4 );
06825             }
06826          }
06827       } else {
06828          // This is a non-disposal case eithe way, so just
06829          // leave the pixels as is, and they will become the new background
06830          // 1: do not dispose
06831          // 0:  not specified.
06832       }
06833
06834       // background is what out is after the undoing of the previou frame;
06835       memcpy( g->background, g->out, 4 * g->w * g->h );
06836    }
06837
06838    // clear my history;
06839    memset( g->history, 0x00, g->w * g->h );        // pixels that were affected previous frame
06840
06841    for (;;) {
06842       int tag = stbi__get8(s);
06843       switch (tag) {
06844          case 0x2C: /* Image Descriptor */
06845          {
06846             stbi__int32 x, y, w, h;
06847             stbi_uc *o;
06848
06849             x = stbi__get16le(s);
06850             y = stbi__get16le(s);
06851             w = stbi__get16le(s);
06852             h = stbi__get16le(s);
06853             if (((x + w) > (g->w)) || ((y + h) > (g->h)))
06854                return stbi__errpuc("bad Image Descriptor", "Corrupt GIF");
06855
06856             g->line_size = g->w * 4;
06857             g->start_x = x * 4;
06858             g->start_y = y * g->line_size;
06859             g->max_x   = g->start_x + w * 4;
06860             g->max_y   = g->start_y + h * g->line_size;
06861             g->cur_x   = g->start_x;
06862             g->cur_y   = g->start_y;
06863
06864             // if the width of the specified rectangle is 0, that means
06865             // we may not see *any* pixels or the image is malformed;
06866             // to make sure this is caught, move the current y down to
06867             // max_y (which is what out_gif_code checks).
06868             if (w == 0)
06869                g->cur_y = g->max_y;
06870
06871             g->lflags = stbi__get8(s);
06872
06873             if (g->lflags & 0x40) {
06874                g->step = 8 * g->line_size; // first interlaced spacing
06875                g->parse = 3;
06876             } else {
06877                g->step = g->line_size;
06878                g->parse = 0;
06879             }
06880
06881             if (g->lflags & 0x80) {
06882                stbi__gif_parse_colortable(s,g->lpal, 2 << (g->lflags & 7), g->eflags & 0x01 ?
      g->transparent : -1);
06883                g->color_table = (stbi_uc *) g->lpal;
06884             } else if (g->flags & 0x80) {
06885                g->color_table = (stbi_uc *) g->pal;
06886             } else
06887                return stbi__errpuc("missing color table", "Corrupt GIF");
06888
06889             o = stbi__process_gif_raster(s, g);
06890             if (!o) return NULL;
06891
06892             // if this was the first frame,
06893             pcount = g->w * g->h;
06894             if (first_frame && (g->bgindex > 0)) {
06895                // if first frame, any pixel not drawn to gets the background color
06896                for (pi = 0; pi < pcount; ++pi) {
06897                   if (g->history[pi] == 0) {
06898                      g->pal[g->bgindex][3] = 255; // just in case it was made transparent, undo that;
      It will be reset next frame if need be;
```

```
06899                      memcpy( &g->out[pi * 4], &g->pal[g->bgindex], 4 );
06900                   }
06901                }
06902             }
06903
06904             return o;
06905          }
06906
06907          case 0x21: // Comment Extension.
06908          {
06909             int len;
06910             int ext = stbi__get8(s);
06911             if (ext == 0xF9) { // Graphic Control Extension.
06912                len = stbi__get8(s);
06913                if (len == 4) {
06914                   g->eflags = stbi__get8(s);
06915                   g->delay = 10 * stbi__get16le(s); // delay - 1/100th of a second, saving as
      1/1000ths.
06916
06917                   // unset old transparent
06918                   if (g->transparent >= 0) {
06919                      g->pal[g->transparent][3] = 255;
06920                   }
06921                   if (g->eflags & 0x01) {
06922                      g->transparent = stbi__get8(s);
06923                      if (g->transparent >= 0) {
06924                         g->pal[g->transparent][3] = 0;
06925                      }
06926                   } else {
06927                      // don't need transparent
06928                      stbi__skip(s, 1);
06929                      g->transparent = -1;
06930                   }
06931                } else {
06932                   stbi__skip(s, len);
06933                   break;
06934                }
06935             }
06936             while ((len = stbi__get8(s)) != 0) {
06937                stbi__skip(s, len);
06938             }
06939             break;
06940          }
06941
06942          case 0x3B: // gif stream termination code
06943             return (stbi_uc *) s; // using '1' causes warning on some compilers
06944
06945          default:
06946             return stbi__errpuc("unknown code", "Corrupt GIF");
06947       }
06948    }
06949 }
06950
06951 static void *stbi__load_gif_main_outofmem(stbi__gif *g, stbi_uc *out, int **delays)
06952 {
06953    STBI_FREE(g->out);
06954    STBI_FREE(g->history);
06955    STBI_FREE(g->background);
06956
06957    if (out) STBI_FREE(out);
06958    if (delays && *delays) STBI_FREE(*delays);
06959    return stbi__errpuc("outofmem", "Out of memory");
06960 }
06961
06962 static void *stbi__load_gif_main(stbi__context *s, int **delays, int *x, int *y, int *z, int *comp,
      int req_comp)
06963 {
06964    if (stbi__gif_test(s)) {
06965       int layers = 0;
06966       stbi_uc *u = 0;
06967       stbi_uc *out = 0;
06968       stbi_uc *two_back = 0;
06969       stbi__gif g;
06970       int stride;
06971       int out_size = 0;
06972       int delays_size = 0;
06973
06974       STBI_NOTUSED(out_size);
06975       STBI_NOTUSED(delays_size);
06976
06977       memset(&g, 0, sizeof(g));
06978       if (delays) {
06979          *delays = 0;
06980       }
06981
06982       do {
06983          u = stbi__gif_load_next(s, &g, comp, req_comp, two_back);
```

```
06984          if (u == (stbi_uc *) s) u = 0;  // end of animated gif marker
06985
06986          if (u) {
06987             *x = g.w;
06988             *y = g.h;
06989             ++layers;
06990             stride = g.w * g.h * 4;
06991
06992             if (out) {
06993                void *tmp = (stbi_uc*) STBI_REALLOC_SIZED( out, out_size, layers * stride );
06994                if (!tmp)
06995                   return stbi__load_gif_main_outofmem(&g, out, delays);
06996                else {
06997                    out = (stbi_uc*) tmp;
06998                    out_size = layers * stride;
06999                }
07000
07001                if (delays) {
07002                   int *new_delays = (int*) STBI_REALLOC_SIZED( *delays, delays_size, sizeof(int) *
      layers );
07003                   if (!new_delays)
07004                      return stbi__load_gif_main_outofmem(&g, out, delays);
07005                   *delays = new_delays;
07006                   delays_size = layers * sizeof(int);
07007                }
07008             } else {
07009                out = (stbi_uc*)stbi__malloc( layers * stride );
07010                if (!out)
07011                   return stbi__load_gif_main_outofmem(&g, out, delays);
07012                out_size = layers * stride;
07013                if (delays) {
07014                   *delays = (int*) stbi__malloc( layers * sizeof(int) );
07015                   if (!*delays)
07016                      return stbi__load_gif_main_outofmem(&g, out, delays);
07017                   delays_size = layers * sizeof(int);
07018                }
07019             }
07020             memcpy( out + ((layers - 1) * stride), u, stride );
07021             if (layers >= 2) {
07022                two_back = out - 2 * stride;
07023             }
07024
07025             if (delays) {
07026                (*delays)[layers - 1U] = g.delay;
07027             }
07028          }
07029       } while (u != 0);
07030
07031       // free temp buffer;
07032       STBI_FREE(g.out);
07033       STBI_FREE(g.history);
07034       STBI_FREE(g.background);
07035
07036       // do the final conversion after loading everything;
07037       if (req_comp && req_comp != 4)
07038          out = stbi__convert_format(out, 4, req_comp, layers * g.w, g.h);
07039
07040       *z = layers;
07041       return out;
07042    } else {
07043       return stbi__errpuc("not GIF", "Image was not as a gif type.");
07044    }
07045 }
07046
07047 static void *stbi__gif_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri)
07048 {
07049    stbi_uc *u = 0;
07050    stbi__gif g;
07051    memset(&g, 0, sizeof(g));
07052    STBI_NOTUSED(ri);
07053
07054    u = stbi__gif_load_next(s, &g, comp, req_comp, 0);
07055    if (u == (stbi_uc *) s) u = 0;  // end of animated gif marker
07056    if (u) {
07057       *x = g.w;
07058       *y = g.h;
07059
07060       // moved conversion to after successful load so that the same
07061       // can be done for multiple frames.
07062       if (req_comp && req_comp != 4)
07063          u = stbi__convert_format(u, 4, req_comp, g.w, g.h);
07064    } else if (g.out) {
07065       // if there was an error and we allocated an image buffer, free it!
07066       STBI_FREE(g.out);
07067    }
07068
```

```
07069    // free buffers needed for multiple frame loading;
07070    STBI_FREE(g.history);
07071    STBI_FREE(g.background);
07072
07073    return u;
07074 }
07075
07076 static int stbi__gif_info(stbi__context *s, int *x, int *y, int *comp)
07077 {
07078    return stbi__gif_info_raw(s,x,y,comp);
07079 }
07080 #endif
07081
07082 // *************************************************************************************************
07083 // Radiance RGBE HDR loader
07084 // originally by Nicolas Schulz
07085 #ifndef STBI_NO_HDR
07086 static int stbi__hdr_test_core(stbi__context *s, const char *signature)
07087 {
07088    int i;
07089    for (i=0; signature[i]; ++i)
07090       if (stbi__get8(s) != signature[i])
07091           return 0;
07092    stbi__rewind(s);
07093    return 1;
07094 }
07095
07096 static int stbi__hdr_test(stbi__context* s)
07097 {
07098    int r = stbi__hdr_test_core(s, "#?RADIANCE\n");
07099    stbi__rewind(s);
07100    if(!r) {
07101        r = stbi__hdr_test_core(s, "#?RGBE\n");
07102        stbi__rewind(s);
07103    }
07104    return r;
07105 }
07106
07107 #define STBI__HDR_BUFLEN  1024
07108 static char *stbi__hdr_gettoken(stbi__context *z, char *buffer)
07109 {
07110    int len=0;
07111    char c = '\0';
07112
07113    c = (char) stbi__get8(z);
07114
07115    while (!stbi__at_eof(z) && c != '\n') {
07116       buffer[len++] = c;
07117       if (len == STBI__HDR_BUFLEN-1) {
07118          // flush to end of line
07119          while (!stbi__at_eof(z) && stbi__get8(z) != '\n')
07120             ;
07121          break;
07122       }
07123       c = (char) stbi__get8(z);
07124    }
07125
07126    buffer[len] = 0;
07127    return buffer;
07128 }
07129
07130 static void stbi__hdr_convert(float *output, stbi_uc *input, int req_comp)
07131 {
07132    if ( input[3] != 0 ) {
07133       float f1;
07134       // Exponent
07135       f1 = (float) ldexp(1.0f, input[3] - (int)(128 + 8));
07136       if (req_comp <= 2)
07137          output[0] = (input[0] + input[1] + input[2]) * f1 / 3;
07138       else {
07139          output[0] = input[0] * f1;
07140          output[1] = input[1] * f1;
07141          output[2] = input[2] * f1;
07142       }
07143       if (req_comp == 2) output[1] = 1;
07144       if (req_comp == 4) output[3] = 1;
07145    } else {
07146       switch (req_comp) {
07147          case 4: output[3] = 1; /* fallthrough */
07148          case 3: output[0] = output[1] = output[2] = 0;
07149                  break;
07150          case 2: output[1] = 1; /* fallthrough */
07151          case 1: output[0] = 0;
07152                  break;
07153       }
07154    }
07155 }
```

```
07156
07157 static float *stbi__hdr_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
      stbi__result_info *ri)
07158 {
07159    char buffer[STBI__HDR_BUFLEN];
07160    char *token;
07161    int valid = 0;
07162    int width, height;
07163    stbi_uc *scanline;
07164    float *hdr_data;
07165    int len;
07166    unsigned char count, value;
07167    int i, j, k, c1,c2, z;
07168    const char *headerToken;
07169    STBI_NOTUSED(ri);
07170
07171    // Check identifier
07172    headerToken = stbi__hdr_gettoken(s,buffer);
07173    if (strcmp(headerToken, "#?RADIANCE") != 0 && strcmp(headerToken, "#?RGBE") != 0)
07174       return stbi__errpf("not HDR", "Corrupt HDR image");
07175
07176    // Parse header
07177    for(;;) {
07178       token = stbi__hdr_gettoken(s,buffer);
07179       if (token[0] == 0) break;
07180       if (strcmp(token, "FORMAT=32-bit_rle_rgbe") == 0) valid = 1;
07181    }
07182
07183    if (!valid)    return stbi__errpf("unsupported format", "Unsupported HDR format");
07184
07185    // Parse width and height
07186    // can’t use sscanf() if we’re not using stdio!
07187    token = stbi__hdr_gettoken(s,buffer);
07188    if (strncmp(token, "-Y ", 3))  return stbi__errpf("unsupported data layout", "Unsupported HDR
      format");
07189    token += 3;
07190    height = (int) strtol(token, &token, 10);
07191    while (*token == ' ') ++token;
07192    if (strncmp(token, "+X ", 3))  return stbi__errpf("unsupported data layout", "Unsupported HDR
      format");
07193    token += 3;
07194    width = (int) strtol(token, NULL, 10);
07195
07196    if (height > STBI_MAX_DIMENSIONS) return stbi__errpf("too large","Very large image (corrupt?)");
07197    if (width > STBI_MAX_DIMENSIONS) return stbi__errpf("too large","Very large image (corrupt?)");
07198
07199    *x = width;
07200    *y = height;
07201
07202    if (comp) *comp = 3;
07203    if (req_comp == 0) req_comp = 3;
07204
07205    if (!stbi__mad4sizes_valid(width, height, req_comp, sizeof(float), 0))
07206       return stbi__errpf("too large", "HDR image is too large");
07207
07208    // Read data
07209    hdr_data = (float *) stbi__malloc_mad4(width, height, req_comp, sizeof(float), 0);
07210    if (!hdr_data)
07211       return stbi__errpf("outofmem", "Out of memory");
07212
07213    // Load image data
07214    // image data is stored as some number of sca
07215    if ( width < 8 || width >= 32768) {
07216       // Read flat data
07217       for (j=0; j < height; ++j) {
07218          for (i=0; i < width; ++i) {
07219             stbi_uc rgbe[4];
07220            main_decode_loop:
07221             stbi__getn(s, rgbe, 4);
07222             stbi__hdr_convert(hdr_data + j * width * req_comp + i * req_comp, rgbe, req_comp);
07223          }
07224       }
07225    } else {
07226       // Read RLE-encoded data
07227       scanline = NULL;
07228
07229       for (j = 0; j < height; ++j) {
07230          c1 = stbi__get8(s);
07231          c2 = stbi__get8(s);
07232          len = stbi__get8(s);
07233          if (c1 != 2 || c2 != 2 || (len & 0x80)) {
07234             // not run-length encoded, so we have to actually use THIS data as a decoded
07235             // pixel (note this can’t be a valid pixel--one of RGB must be >= 128)
07236             stbi_uc rgbe[4];
07237             rgbe[0] = (stbi_uc) c1;
07238             rgbe[1] = (stbi_uc) c2;
07239             rgbe[2] = (stbi_uc) len;
```

```
07240            rgbe[3] = (stbi_uc) stbi__get8(s);
07241            stbi__hdr_convert(hdr_data, rgbe, req_comp);
07242            i = 1;
07243            j = 0;
07244            STBI_FREE(scanline);
07245            goto main_decode_loop; // yes, this makes no sense
07246         }
07247         len <<= 8;
07248         len |= stbi__get8(s);
07249         if (len != width) { STBI_FREE(hdr_data); STBI_FREE(scanline); return stbi__errpf("invalid
      decoded scanline length", "corrupt HDR"); }
07250         if (scanline == NULL) {
07251            scanline = (stbi_uc *) stbi__malloc_mad2(width, 4, 0);
07252            if (!scanline) {
07253               STBI_FREE(hdr_data);
07254               return stbi__errpf("outofmem", "Out of memory");
07255            }
07256         }
07257
07258         for (k = 0; k < 4; ++k) {
07259            int nleft;
07260            i = 0;
07261            while ((nleft = width - i) > 0) {
07262               count = stbi__get8(s);
07263               if (count > 128) {
07264                  // Run
07265                  value = stbi__get8(s);
07266                  count -= 128;
07267                  if ((count == 0) || (count > nleft)) { STBI_FREE(hdr_data); STBI_FREE(scanline);
      return stbi__errpf("corrupt", "bad RLE data in HDR"); }
07268                  for (z = 0; z < count; ++z)
07269                     scanline[i++ * 4 + k] = value;
07270               } else {
07271                  // Dump
07272                  if ((count == 0) || (count > nleft)) { STBI_FREE(hdr_data); STBI_FREE(scanline);
      return stbi__errpf("corrupt", "bad RLE data in HDR"); }
07273                  for (z = 0; z < count; ++z)
07274                     scanline[i++ * 4 + k] = stbi__get8(s);
07275               }
07276            }
07277         }
07278         for (i=0; i < width; ++i)
07279            stbi__hdr_convert(hdr_data+(j*width + i)*req_comp, scanline + i*4, req_comp);
07280      }
07281      if (scanline)
07282         STBI_FREE(scanline);
07283   }
07284
07285   return hdr_data;
07286 }
07287
07288 static int stbi__hdr_info(stbi__context *s, int *x, int *y, int *comp)
07289 {
07290   char buffer[STBI__HDR_BUFLEN];
07291   char *token;
07292   int valid = 0;
07293   int dummy;
07294
07295   if (!x) x = &dummy;
07296   if (!y) y = &dummy;
07297   if (!comp) comp = &dummy;
07298
07299   if (stbi__hdr_test(s) == 0) {
07300       stbi__rewind( s );
07301       return 0;
07302   }
07303
07304   for(;;) {
07305      token = stbi__hdr_gettoken(s,buffer);
07306      if (token[0] == 0) break;
07307      if (strcmp(token, "FORMAT=32-bit_rle_rgbe") == 0) valid = 1;
07308   }
07309
07310   if (!valid) {
07311       stbi__rewind( s );
07312       return 0;
07313   }
07314   token = stbi__hdr_gettoken(s,buffer);
07315   if (strncmp(token, "-Y ", 3)) {
07316       stbi__rewind( s );
07317       return 0;
07318   }
07319   token += 3;
07320   *y = (int) strtol(token, &token, 10);
07321   while (*token == ' ') ++token;
07322   if (strncmp(token, "+X ", 3)) {
07323       stbi__rewind( s );
```

```
07324         return 0;
07325     }
07326     token += 3;
07327     *x = (int) strtol(token, NULL, 10);
07328     *comp = 3;
07329     return 1;
07330 }
07331 #endif // STBI_NO_HDR
07332
07333 #ifndef STBI_NO_BMP
07334 static int stbi__bmp_info(stbi__context *s, int *x, int *y, int *comp)
07335 {
07336    void *p;
07337    stbi__bmp_data info;
07338
07339    info.all_a = 255;
07340    p = stbi__bmp_parse_header(s, &info);
07341    if (p == NULL) {
07342       stbi__rewind( s );
07343       return 0;
07344    }
07345    if (x) *x = s->img_x;
07346    if (y) *y = s->img_y;
07347    if (comp) {
07348       if (info.bpp == 24 && info.ma == 0xff000000)
07349          *comp = 3;
07350       else
07351          *comp = info.ma ? 4 : 3;
07352    }
07353    return 1;
07354 }
07355 #endif
07356
07357 #ifndef STBI_NO_PSD
07358 static int stbi__psd_info(stbi__context *s, int *x, int *y, int *comp)
07359 {
07360    int channelCount, dummy, depth;
07361    if (!x) x = &dummy;
07362    if (!y) y = &dummy;
07363    if (!comp) comp = &dummy;
07364    if (stbi__get32be(s) != 0x38425053) {
07365        stbi__rewind( s );
07366        return 0;
07367    }
07368    if (stbi__get16be(s) != 1) {
07369        stbi__rewind( s );
07370        return 0;
07371    }
07372    stbi__skip(s, 6);
07373    channelCount = stbi__get16be(s);
07374    if (channelCount < 0 || channelCount > 16) {
07375        stbi__rewind( s );
07376        return 0;
07377    }
07378    *y = stbi__get32be(s);
07379    *x = stbi__get32be(s);
07380    depth = stbi__get16be(s);
07381    if (depth != 8 && depth != 16) {
07382        stbi__rewind( s );
07383        return 0;
07384    }
07385    if (stbi__get16be(s) != 3) {
07386        stbi__rewind( s );
07387        return 0;
07388    }
07389    *comp = 4;
07390    return 1;
07391 }
07392
07393 static int stbi__psd_is16(stbi__context *s)
07394 {
07395    int channelCount, depth;
07396    if (stbi__get32be(s) != 0x38425053) {
07397        stbi__rewind( s );
07398        return 0;
07399    }
07400    if (stbi__get16be(s) != 1) {
07401        stbi__rewind( s );
07402        return 0;
07403    }
07404    stbi__skip(s, 6);
07405    channelCount = stbi__get16be(s);
07406    if (channelCount < 0 || channelCount > 16) {
07407        stbi__rewind( s );
07408        return 0;
07409    }
07410    STBI_NOTUSED(stbi__get32be(s));
```

```
07411    STBI_NOTUSED(stbi__get32be(s));
07412    depth = stbi__get16be(s);
07413    if (depth != 16) {
07414        stbi__rewind( s );
07415        return 0;
07416    }
07417    return 1;
07418 }
07419 #endif
07420
07421 #ifndef STBI_NO_PIC
07422 static int stbi__pic_info(stbi__context *s, int *x, int *y, int *comp)
07423 {
07424    int act_comp=0,num_packets=0,chained,dummy;
07425    stbi__pic_packet packets[10];
07426
07427    if (!x) x = &dummy;
07428    if (!y) y = &dummy;
07429    if (!comp) comp = &dummy;
07430
07431    if (!stbi__pic_is4(s,"\x53\x80\xF6\x34")) {
07432       stbi__rewind(s);
07433       return 0;
07434    }
07435
07436    stbi__skip(s, 88);
07437
07438    *x = stbi__get16be(s);
07439    *y = stbi__get16be(s);
07440    if (stbi__at_eof(s)) {
07441       stbi__rewind( s);
07442       return 0;
07443    }
07444    if ( (*x) != 0 && (1 << 28) / (*x) < (*y)) {
07445       stbi__rewind( s );
07446       return 0;
07447    }
07448
07449    stbi__skip(s, 8);
07450
07451    do {
07452       stbi__pic_packet *packet;
07453
07454       if (num_packets==sizeof(packets)/sizeof(packets[0]))
07455          return 0;
07456
07457       packet = &packets[num_packets++];
07458       chained = stbi__get8(s);
07459       packet->size    = stbi__get8(s);
07460       packet->type    = stbi__get8(s);
07461       packet->channel = stbi__get8(s);
07462       act_comp |= packet->channel;
07463
07464       if (stbi__at_eof(s)) {
07465           stbi__rewind( s );
07466           return 0;
07467       }
07468       if (packet->size != 8) {
07469           stbi__rewind( s );
07470           return 0;
07471       }
07472    } while (chained);
07473
07474    *comp = (act_comp & 0x10 ? 4 : 3);
07475
07476    return 1;
07477 }
07478 #endif
07479
07480 // *************************************************************************************************
07481 // Portable Gray Map and Portable Pixel Map loader
07482 // by Ken Miller
07483 //
07484 // PGM: http://netpbm.sourceforge.net/doc/pgm.html
07485 // PPM: http://netpbm.sourceforge.net/doc/ppm.html
07486 //
07487 // Known limitations:
07488 //    Does not support comments in the header section
07489 //    Does not support ASCII image data (formats P2 and P3)
07490
07491 #ifndef STBI_NO_PNM
07492
07493 static int      stbi__pnm_test(stbi__context *s)
07494 {
07495    char p, t;
07496    p = (char) stbi__get8(s);
07497    t = (char) stbi__get8(s);
```

```
07498    if (p != 'P' || (t != '5' && t != '6')) {
07499        stbi__rewind( s );
07500        return 0;
07501    }
07502    return 1;
07503 }
07504
07505 static void *stbi__pnm_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
     stbi__result_info *ri)
07506 {
07507    stbi_uc *out;
07508    STBI_NOTUSED(ri);
07509
07510    ri->bits_per_channel = stbi__pnm_info(s, (int *)&s->img_x, (int *)&s->img_y, (int *)&s->img_n);
07511    if (ri->bits_per_channel == 0)
07512       return 0;
07513
07514    if (s->img_y > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
07515    if (s->img_x > STBI_MAX_DIMENSIONS) return stbi__errpuc("too large","Very large image (corrupt?)");
07516
07517    *x = s->img_x;
07518    *y = s->img_y;
07519    if (comp) *comp = s->img_n;
07520
07521    if (!stbi__mad4sizes_valid(s->img_n, s->img_x, s->img_y, ri->bits_per_channel / 8, 0))
07522       return stbi__errpuc("too large", "PNM too large");
07523
07524    out = (stbi_uc *) stbi__malloc_mad4(s->img_n, s->img_x, s->img_y, ri->bits_per_channel / 8, 0);
07525    if (!out) return stbi__errpuc("outofmem", "Out of memory");
07526    if (!stbi__getn(s, out, s->img_n * s->img_x * s->img_y * (ri->bits_per_channel / 8))) {
07527       STBI_FREE(out);
07528       return stbi__errpuc("bad PNM", "PNM file truncated");
07529    }
07530
07531    if (req_comp && req_comp != s->img_n) {
07532       if (ri->bits_per_channel == 16) {
07533          out = (stbi_uc *) stbi__convert_format16((stbi__uint16 *) out, s->img_n, req_comp, s->img_x,
     s->img_y);
07534       } else {
07535          out = stbi__convert_format(out, s->img_n, req_comp, s->img_x, s->img_y);
07536       }
07537       if (out == NULL) return out; // stbi__convert_format frees input on failure
07538    }
07539    return out;
07540 }
07541
07542 static int      stbi__pnm_isspace(char c)
07543 {
07544    return c == ' ' || c == '\t' || c == '\n' || c == '\v' || c == '\f' || c == '\r';
07545 }
07546
07547 static void     stbi__pnm_skip_whitespace(stbi__context *s, char *c)
07548 {
07549    for (;;) {
07550       while (!stbi__at_eof(s) && stbi__pnm_isspace(*c))
07551          *c = (char) stbi__get8(s);
07552
07553       if (stbi__at_eof(s) || *c != '#')
07554          break;
07555
07556       while (!stbi__at_eof(s) && *c != '\n' && *c != '\r' )
07557          *c = (char) stbi__get8(s);
07558    }
07559 }
07560
07561 static int      stbi__pnm_isdigit(char c)
07562 {
07563    return c >= '0' && c <= '9';
07564 }
07565
07566 static int      stbi__pnm_getinteger(stbi__context *s, char *c)
07567 {
07568    int value = 0;
07569
07570    while (!stbi__at_eof(s) && stbi__pnm_isdigit(*c)) {
07571       value = value*10 + (*c - '0');
07572       *c = (char) stbi__get8(s);
07573       if((value > 214748364) || (value == 214748364 && *c > '7'))
07574           return stbi__err("integer parse overflow", "Parsing an integer in the PPM header overflowed
     a 32-bit int");
07575    }
07576
07577    return value;
07578 }
07579
07580 static int      stbi__pnm_info(stbi__context *s, int *x, int *y, int *comp)
07581 {
```

```
07582    int maxv, dummy;
07583    char c, p, t;
07584
07585    if (!x) x = &dummy;
07586    if (!y) y = &dummy;
07587    if (!comp) comp = &dummy;
07588
07589    stbi__rewind(s);
07590
07591    // Get identifier
07592    p = (char) stbi__get8(s);
07593    t = (char) stbi__get8(s);
07594    if (p != 'P' || (t != '5' && t != '6')) {
07595        stbi__rewind(s);
07596        return 0;
07597    }
07598
07599    *comp = (t == '6') ? 3 : 1;  // '5' is 1-component .pgm; '6' is 3-component .ppm
07600
07601    c = (char) stbi__get8(s);
07602    stbi__pnm_skip_whitespace(s, &c);
07603
07604    *x = stbi__pnm_getinteger(s, &c); // read width
07605    if(*x == 0)
07606        return stbi__err("invalid width", "PPM image header had zero or overflowing width");
07607    stbi__pnm_skip_whitespace(s, &c);
07608
07609    *y = stbi__pnm_getinteger(s, &c); // read height
07610    if (*y == 0)
07611        return stbi__err("invalid width", "PPM image header had zero or overflowing width");
07612    stbi__pnm_skip_whitespace(s, &c);
07613
07614    maxv = stbi__pnm_getinteger(s, &c);  // read max value
07615    if (maxv > 65535)
07616       return stbi__err("max value > 65535", "PPM image supports only 8-bit and 16-bit images");
07617    else if (maxv > 255)
07618       return 16;
07619    else
07620       return 8;
07621 }
07622
07623 static int stbi__pnm_is16(stbi__context *s)
07624 {
07625    if (stbi__pnm_info(s, NULL, NULL, NULL) == 16)
07626        return 1;
07627    return 0;
07628 }
07629 #endif
07630
07631 static int stbi__info_main(stbi__context *s, int *x, int *y, int *comp)
07632 {
07633    #ifndef STBI_NO_JPEG
07634    if (stbi__jpeg_info(s, x, y, comp)) return 1;
07635    #endif
07636
07637    #ifndef STBI_NO_PNG
07638    if (stbi__png_info(s, x, y, comp))  return 1;
07639    #endif
07640
07641    #ifndef STBI_NO_GIF
07642    if (stbi__gif_info(s, x, y, comp))  return 1;
07643    #endif
07644
07645    #ifndef STBI_NO_BMP
07646    if (stbi__bmp_info(s, x, y, comp))  return 1;
07647    #endif
07648
07649    #ifndef STBI_NO_PSD
07650    if (stbi__psd_info(s, x, y, comp))  return 1;
07651    #endif
07652
07653    #ifndef STBI_NO_PIC
07654    if (stbi__pic_info(s, x, y, comp))  return 1;
07655    #endif
07656
07657    #ifndef STBI_NO_PNM
07658    if (stbi__pnm_info(s, x, y, comp))  return 1;
07659    #endif
07660
07661    #ifndef STBI_NO_HDR
07662    if (stbi__hdr_info(s, x, y, comp))  return 1;
07663    #endif
07664
07665    // test tga last because it's a crappy test!
07666    #ifndef STBI_NO_TGA
07667    if (stbi__tga_info(s, x, y, comp))
07668        return 1;
```

```
07669    #endif
07670    return stbi__err("unknown image type", "Image not of any known type, or corrupt");
07671 }
07672
07673 static int stbi__is_16_main(stbi__context *s)
07674 {
07675    #ifndef STBI_NO_PNG
07676    if (stbi__png_is16(s))  return 1;
07677    #endif
07678
07679    #ifndef STBI_NO_PSD
07680    if (stbi__psd_is16(s))  return 1;
07681    #endif
07682
07683    #ifndef STBI_NO_PNM
07684    if (stbi__pnm_is16(s))  return 1;
07685    #endif
07686    return 0;
07687 }
07688
07689 #ifndef STBI_NO_STDIO
07690 STBIDEF int stbi_info(char const *filename, int *x, int *y, int *comp)
07691 {
07692     FILE *f = stbi__fopen(filename, "rb");
07693     int result;
07694     if (!f) return stbi__err("can't fopen", "Unable to open file");
07695     result = stbi_info_from_file(f, x, y, comp);
07696     fclose(f);
07697     return result;
07698 }
07699
07700 STBIDEF int stbi_info_from_file(FILE *f, int *x, int *y, int *comp)
07701 {
07702    int r;
07703    stbi__context s;
07704    long pos = ftell(f);
07705    stbi__start_file(&s, f);
07706    r = stbi__info_main(&s,x,y,comp);
07707    fseek(f,pos,SEEK_SET);
07708    return r;
07709 }
07710
07711 STBIDEF int stbi_is_16_bit(char const *filename)
07712 {
07713     FILE *f = stbi__fopen(filename, "rb");
07714     int result;
07715     if (!f) return stbi__err("can't fopen", "Unable to open file");
07716     result = stbi_is_16_bit_from_file(f);
07717     fclose(f);
07718     return result;
07719 }
07720
07721 STBIDEF int stbi_is_16_bit_from_file(FILE *f)
07722 {
07723    int r;
07724    stbi__context s;
07725    long pos = ftell(f);
07726    stbi__start_file(&s, f);
07727    r = stbi__is_16_main(&s);
07728    fseek(f,pos,SEEK_SET);
07729    return r;
07730 }
07731 #endif // !STBI_NO_STDIO
07732
07733 STBIDEF int stbi_info_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp)
07734 {
07735    stbi__context s;
07736    stbi__start_mem(&s,buffer,len);
07737    return stbi__info_main(&s,x,y,comp);
07738 }
07739
07740 STBIDEF int stbi_info_from_callbacks(stbi_io_callbacks const *c, void *user, int *x, int *y, int
      *comp)
07741 {
07742    stbi__context s;
07743    stbi__start_callbacks(&s, (stbi_io_callbacks *) c, user);
07744    return stbi__info_main(&s,x,y,comp);
07745 }
07746
07747 STBIDEF int stbi_is_16_bit_from_memory(stbi_uc const *buffer, int len)
07748 {
07749    stbi__context s;
07750    stbi__start_mem(&s,buffer,len);
07751    return stbi__is_16_main(&s);
07752 }
07753
07754 STBIDEF int stbi_is_16_bit_from_callbacks(stbi_io_callbacks const *c, void *user)
```

```
07755 {
07756    stbi__context s;
07757    stbi__start_callbacks(&s, (stbi_io_callbacks *) c, user);
07758    return stbi__is_16_main(&s);
07759 }
07760
07761 #endif // STB_IMAGE_IMPLEMENTATION
07762
07763 /*
07764    revision history:
07765       2.20  (2019-02-07) support utf8 filenames in Windows; fix warnings and platform ifdefs
07766       2.19  (2018-02-11) fix warning
07767       2.18  (2018-01-30) fix warnings
07768       2.17  (2018-01-29) change sbti__shiftsigned to avoid clang -O2 bug
07769                          1-bit BMP
07770                          *_is_16_bit api
07771                          avoid warnings
07772       2.16  (2017-07-23) all functions have 16-bit variants;
07773                          STBI_NO_STDIO works again;
07774                          compilation fixes;
07775                          fix rounding in unpremultiply;
07776                          optimize vertical flip;
07777                          disable raw_len validation;
07778                          documentation fixes
07779       2.15  (2017-03-18) fix png-1,2,4 bug; now all Imagenet JPGs decode;
07780                          warning fixes; disable run-time SSE detection on gcc;
07781                          uniform handling of optional "return" values;
07782                          thread-safe initialization of zlib tables
07783       2.14  (2017-03-03) remove deprecated STBI_JPEG_OLD; fixes for Imagenet JPGs
07784       2.13  (2016-11-29) add 16-bit API, only supported for PNG right now
07785       2.12  (2016-04-02) fix typo in 2.11 PSD fix that caused crashes
07786       2.11  (2016-04-02) allocate large structures on the stack
07787                          remove white matting for transparent PSD
07788                          fix reported channel count for PNG & BMP
07789                          re-enable SSE2 in non-gcc 64-bit
07790                          support RGB-formatted JPEG
07791                          read 16-bit PNGs (only as 8-bit)
07792       2.10  (2016-01-22) avoid warning introduced in 2.09 by STBI_REALLOC_SIZED
07793       2.09  (2016-01-16) allow comments in PNM files
07794                          16-bit-per-pixel TGA (not bit-per-component)
07795                          info() for TGA could break due to .hdr handling
07796                          info() for BMP to shares code instead of sloppy parse
07797                          can use STBI_REALLOC_SIZED if allocator doesn't support realloc
07798                          code cleanup
07799       2.08  (2015-09-13) fix to 2.07 cleanup, reading RGB PSD as RGBA
07800       2.07  (2015-09-13) fix compiler warnings
07801                          partial animated GIF support
07802                          limited 16-bpc PSD support
07803                          #ifdef unused functions
07804                          bug with < 92 byte PIC,PNM,HDR,TGA
07805       2.06  (2015-04-19) fix bug where PSD returns wrong '*comp' value
07806       2.05  (2015-04-19) fix bug in progressive JPEG handling, fix warning
07807       2.04  (2015-04-15) try to re-enable SIMD on MinGW 64-bit
07808       2.03  (2015-04-12) extra corruption checking (mmozeiko)
07809                          stbi_set_flip_vertically_on_load (nguillemot)
07810                          fix NEON support; fix mingw support
07811       2.02  (2015-01-19) fix incorrect assert, fix warning
07812       2.01  (2015-01-17) fix various warnings; suppress SIMD on gcc 32-bit without -msse2
07813       2.00b (2014-12-25) fix STBI_MALLOC in progressive JPEG
07814       2.00  (2014-12-25) optimize JPG, including x86 SSE2 & NEON SIMD (ryg)
07815                          progressive JPEG (stb)
07816                          PGM/PPM support (Ken Miller)
07817                          STBI_MALLOC,STBI_REALLOC,STBI_FREE
07818                          GIF bugfix -- seemingly never worked
07819                          STBI_NO_*, STBI_ONLY_*
07820       1.48  (2014-12-14) fix incorrectly-named assert()
07821       1.47  (2014-12-14) 1/2/4-bit PNG support, both direct and paletted (Omar Cornut & stb)
07822                          optimize PNG (ryg)
07823                          fix bug in interlaced PNG with user-specified channel count (stb)
07824       1.46  (2014-08-26)
07825                fix broken tRNS chunk (colorkey-style transparency) in non-paletted PNG
07826       1.45  (2014-08-16)
07827                fix MSVC-ARM internal compiler error by wrapping malloc
07828       1.44  (2014-08-07)
07829                various warning fixes from Ronny Chevalier
07830       1.43  (2014-07-15)
07831                fix MSVC-only compiler problem in code changed in 1.42
07832       1.42  (2014-07-09)
07833                don't define _CRT_SECURE_NO_WARNINGS (affects user code)
07834                fixes to stbi__cleanup_jpeg path
07835                added STBI_ASSERT to avoid requiring assert.h
07836       1.41  (2014-06-25)
07837                fix search&replace from 1.36 that messed up comments/error messages
07838       1.40  (2014-06-22)
07839                fix gcc struct-initialization warning
07840       1.39  (2014-06-15)
07841                fix to TGA optimization when req_comp != number of components in TGA;
```

```
07842                fix to GIF loading because BMP wasn't rewinding (whoops, no GIFs in my test suite)
07843                add support for BMP version 5 (more ignored fields)
07844        1.38  (2014-06-06)
07845                suppress MSVC warnings on integer casts truncating values
07846                fix accidental rename of 'skip' field of I/O
07847        1.37  (2014-06-04)
07848                remove duplicate typedef
07849        1.36  (2014-06-03)
07850                convert to header file single-file library
07851                if de-iphone isn't set, load iphone images color-swapped instead of returning NULL
07852        1.35  (2014-05-27)
07853                various warnings
07854                fix broken STBI_SIMD path
07855                fix bug where stbi_load_from_file no longer left file pointer in correct place
07856                fix broken non-easy path for 32-bit BMP (possibly never used)
07857                TGA optimization by Arseny Kapoulkine
07858        1.34  (unknown)
07859                use STBI_NOTUSED in stbi__resample_row_generic(), fix one more leak in tga failure case
07860        1.33  (2011-07-14)
07861                make stbi_is_hdr work in STBI_NO_HDR (as specified), minor compiler-friendly
      improvements
07862        1.32  (2011-07-13)
07863                support for "info" function for all supported filetypes (SpartanJ)
07864        1.31  (2011-06-20)
07865                a few more leak fixes, bug in PNG handling (SpartanJ)
07866        1.30  (2011-06-11)
07867                added ability to load files via callbacks to accomidate custom input streams (Ben
      Wenger)
07868                removed deprecated format-specific test/load functions
07869                removed support for installable file formats (stbi_loader) -- would have been broken for
      IO callbacks anyway
07870                error cases in bmp and tga give messages and don't leak (Raymond Barbiero, grisha)
07871                fix inefficiency in decoding 32-bit BMP (David Woo)
07872        1.29  (2010-08-16)
07873                various warning fixes from Aurelien Pocheville
07874        1.28  (2010-08-01)
07875                fix bug in GIF palette transparency (SpartanJ)
07876        1.27  (2010-08-01)
07877                cast-to-stbi_uc to fix warnings
07878        1.26  (2010-07-24)
07879                fix bug in file buffering for PNG reported by SpartanJ
07880        1.25  (2010-07-17)
07881                refix trans_data warning (Won Chun)
07882        1.24  (2010-07-12)
07883                perf improvements reading from files on platforms with lock-heavy fgetc()
07884                minor perf improvements for jpeg
07885                deprecated type-specific functions so we'll get feedback if they're needed
07886                attempt to fix trans_data warning (Won Chun)
07887        1.23    fixed bug in iPhone support
07888        1.22  (2010-07-10)
07889                removed image *writing* support
07890                stbi_info support from Jetro Lauha
07891                GIF support from Jean-Marc Lienher
07892                iPhone PNG-extensions from James Brown
07893                warning-fixes from Nicolas Schulz and Janez Zemva (i.stbi__err. Janez (U+017D)emva)
07894        1.21    fix use of 'stbi_uc' in header (reported by jon blow)
07895        1.20    added support for Softimage PIC, by Tom Seddon
07896        1.19    bug in interlaced PNG corruption check (found by ryg)
07897        1.18  (2008-08-02)
07898                fix a threading bug (local mutable static)
07899        1.17    support interlaced PNG
07900        1.16    major bugfix - stbi__convert_format converted one too many pixels
07901        1.15    initialize some fields for thread safety
07902        1.14    fix threadsafe conversion bug
07903                header-file-only version (#define STBI_HEADER_FILE_ONLY before including)
07904        1.13    threadsafe
07905        1.12    const qualifiers in the API
07906        1.11    Support installable IDCT, colorspace conversion routines
07907        1.10    Fixes for 64-bit (don't use "unsigned long")
07908                optimized upsampling by Fabian "ryg" Giesen
07909        1.09    Fix format-conversion for PSD code (bad global variables!)
07910        1.08    Thatcher Ulrich's PSD code integrated by Nicolas Schulz
07911        1.07    attempt to fix C++ warning/errors again
07912        1.06    attempt to fix C++ warning/errors again
07913        1.05    fix TGA loading to return correct *comp and use good luminance calc
07914        1.04    default float alpha is 1, not 255; use 'void *' for stbi_image_free
07915        1.03    bugfixes to STBI_NO_STDIO, STBI_NO_HDR
07916        1.02    support for (subset of) HDR files, float interface for preferred access to them
07917        1.01    fix bug: possible bug in handling right-side up bmps... not sure
07918                fix bug: the stbi__bmp_load() and stbi__tga_load() functions didn't work at all
07919        1.00    interface to zlib that skips zlib header
07920        0.99    correct handling of alpha in palette
07921        0.98    TGA loader by lonesock; dynamically add loaders (untested)
07922        0.97    jpeg errors on too large a file; also catch another malloc failure
07923        0.96    fix detection of invalid v value - particleman@mollyrocket forum
07924        0.95    during header scan, seek to markers in case of padding
07925        0.94    STBI_NO_STDIO to disable stdio usage; rename all #defines the same
```

```
07926       0.93    handle jpegtran output; verbose errors
07927       0.92    read 4,8,16,24,32-bit BMP files of several formats
07928       0.91    output 24-bit Windows 3.0 BMP files
07929       0.90    fix a few more warnings; bump version number to approach 1.0
07930       0.61    bugfixes due to Marc LeBlanc, Christopher Lloyd
07931       0.60    fix compiling as c++
07932       0.59    fix warnings: merge Dave Moore's -Wall fixes
07933       0.58    fix bug: zlib uncompressed mode len/nlen was wrong endian
07934       0.57    fix bug: jpg last huffman symbol before marker was >9 bits but less than 16 available
07935       0.56    fix bug: zlib uncompressed mode len vs. nlen
07936       0.55    fix bug: restart_interval not initialized to 0
07937       0.54    allow NULL for 'int *comp'
07938       0.53    fix bug in png 3->4; speedup png decoding
07939       0.52    png handles req_comp=3,4 directly; minor cleanup; jpeg comments
07940       0.51    obey req_comp requests, 1-component jpegs return as 1-component,
07941               on 'test' only check type, not whether we support this variant
07942       0.50  (2006-11-19)
07943               first released version
07944 */
07945
07946
07947 /*
07948 ------------------------------------------------------------------------------
07949 This software is available under 2 licenses -- choose whichever you prefer.
07950 ------------------------------------------------------------------------------
07951 ALTERNATIVE A - MIT License
07952 Copyright (c) 2017 Sean Barrett
07953 Permission is hereby granted, free of charge, to any person obtaining a copy of
07954 this software and associated documentation files (the "Software"), to deal in
07955 the Software without restriction, including without limitation the rights to
07956 use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
07957 of the Software, and to permit persons to whom the Software is furnished to do
07958 so, subject to the following conditions:
07959 The above copyright notice and this permission notice shall be included in all
07960 copies or substantial portions of the Software.
07961 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
07962 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
07963 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
07964 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
07965 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
07966 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
07967 SOFTWARE.
07968 ------------------------------------------------------------------------------
07969 ALTERNATIVE B - Public Domain (www.unlicense.org)
07970 This is free and unencumbered software released into the public domain.
07971 Anyone is free to copy, modify, publish, use, compile, sell, or distribute this
07972 software, either in source code form or as a compiled binary, for any purpose,
07973 commercial or non-commercial, and by any means.
07974 In jurisdictions that recognize copyright laws, the author or authors of this
07975 software dedicate any and all copyright interest in the software to the public
07976 domain. We make this dedication for the benefit of the public at large and to
07977 the detriment of our heirs and successors. We intend this dedication to be an
07978 overt act of relinquishment in perpetuity of all present and future rights to
07979 this software under copyright law.
07980 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
07981 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
07982 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
07983 AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
07984 ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
07985 WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
07986 ------------------------------------------------------------------------------
07987 */
```

5.26 stb/stb_image_write.h File Reference

`#include <stdlib.h>`

Macros

- #define STBIWDEF extern

Typedefs

- typedef void stbi_write_func(void *context, void *data, int size)

Functions

- STBIWDEF int stbi_write_png (char const *filename, int w, int h, int comp, const void *data, int stride_in_↩ bytes)
- STBIWDEF int stbi_write_bmp (char const *filename, int w, int h, int comp, const void *data)
- STBIWDEF int stbi_write_tga (char const *filename, int w, int h, int comp, const void *data)
- STBIWDEF int stbi_write_hdr (char const *filename, int w, int h, int comp, const float *data)
- STBIWDEF int stbi_write_jpg (char const *filename, int x, int y, int comp, const void *data, int quality)
- STBIWDEF int stbi_write_png_to_func (stbi_write_func *func, void *context, int w, int h, int comp, const void *data, int stride_in_bytes)
- STBIWDEF int stbi_write_bmp_to_func (stbi_write_func *func, void *context, int w, int h, int comp, const void *data)
- STBIWDEF int stbi_write_tga_to_func (stbi_write_func *func, void *context, int w, int h, int comp, const void *data)
- STBIWDEF int stbi_write_hdr_to_func (stbi_write_func *func, void *context, int w, int h, int comp, const float *data)
- STBIWDEF int stbi_write_jpg_to_func (stbi_write_func *func, void *context, int x, int y, int comp, const void *data, int quality)
- STBIWDEF void stbi_flip_vertically_on_write (int flip_boolean)

Variables

- STBIWDEF int stbi_write_tga_with_rle
- STBIWDEF int stbi_write_png_compression_level
- STBIWDEF int stbi_write_force_png_filter

5.26.1 Macro Definition Documentation

5.26.1.1 STBIWDEF

```
#define STBIWDEF extern
```

5.26.2 Typedef Documentation

5.26.2.1 stbi_write_func

```
typedef void stbi_write_func(void *context, void *data, int size)
```

5.26.3 Function Documentation

5.26.3.1 stbi_flip_vertically_on_write()

```
STBIWDEF void stbi_flip_vertically_on_write (
            int flip_boolean )
```

5.26.3.2 stbi_write_bmp()

```
STBIWDEF int stbi_write_bmp (
            char const * filename,
            int w,
            int h,
            int comp,
            const void * data )
```

5.26.3.3 stbi_write_bmp_to_func()

```
STBIWDEF int stbi_write_bmp_to_func (
            stbi_write_func * func,
            void * context,
            int w,
            int h,
            int comp,
            const void * data )
```

5.26.3.4 stbi_write_hdr()

```
STBIWDEF int stbi_write_hdr (
            char const * filename,
            int w,
            int h,
            int comp,
            const float * data )
```

5.26.3.5 stbi_write_hdr_to_func()

```
STBIWDEF int stbi_write_hdr_to_func (
            stbi_write_func * func,
            void * context,
            int w,
            int h,
            int comp,
            const float * data )
```

5.26.3.6 stbi_write_jpg()

```
STBIWDEF int stbi_write_jpg (
            char const * filename,
            int x,
            int y,
            int comp,
            const void * data,
            int quality )
```

5.26.3.7 stbi_write_jpg_to_func()

```
STBIWDEF int stbi_write_jpg_to_func (
            stbi_write_func * func,
            void * context,
            int x,
            int y,
            int comp,
            const void * data,
            int quality )
```

5.26.3.8 stbi_write_png()

```
STBIWDEF int stbi_write_png (
            char const * filename,
            int w,
            int h,
            int comp,
            const void * data,
            int stride_in_bytes )
```

5.26.3.9 stbi_write_png_to_func()

```
STBIWDEF int stbi_write_png_to_func (
            stbi_write_func * func,
            void * context,
            int w,
            int h,
            int comp,
            const void * data,
            int stride_in_bytes )
```

5.26.3.10 stbi_write_tga()

```
STBIWDEF int stbi_write_tga (
            char const * filename,
            int w,
            int h,
            int comp,
            const void * data )
```

5.26.3.11 stbi_write_tga_to_func()

```
STBIWDEF int stbi_write_tga_to_func (
            stbi_write_func * func,
            void * context,
            int w,
            int h,
            int comp,
            const void * data )
```

5.26.4 Variable Documentation

5.26.4.1 stbi_write_force_png_filter

```
STBIWDEF int stbi_write_force_png_filter
```

5.26.4.2 stbi_write_png_compression_level

```
STBIWDEF int stbi_write_png_compression_level
```

5.26.4.3 stbi_write_tga_with_rle

```
STBIWDEF int stbi_write_tga_with_rle
```

5.27 stb_image_write.h

Go to the documentation of this file.

```
00001 /* stb_image_write - v1.16 - public domain - http://nothings.org/stb
00002    writes out PNG/BMP/TGA/JPEG/HDR images to C stdio - Sean Barrett 2010-2015
00003                                      no warranty implied; use at your own risk
00004
00005    Before #including,
00006
00007        #define STB_IMAGE_WRITE_IMPLEMENTATION
00008
00009    in the file that you want to have the implementation.
00010
00011    Will probably not work correctly with strict-aliasing optimizations.
00012
00013 ABOUT:
00014
00015    This header file is a library for writing images to C stdio or a callback.
00016
00017    The PNG output is not optimal; it is 20-50% larger than the file
00018    written by a decent optimizing implementation; though providing a custom
00019    zlib compress function (see STBIW_ZLIB_COMPRESS) can mitigate that.
00020    This library is designed for source code compactness and simplicity,
00021    not optimal image file size or run-time performance.
00022
00023 BUILDING:
00024
00025    You can #define STBIW_ASSERT(x) before the #include to avoid using assert.h.
00026    You can #define STBIW_MALLOC(), STBIW_REALLOC(), and STBIW_FREE() to replace
00027    malloc,realloc,free.
00028    You can #define STBIW_MEMMOVE() to replace memmove()
00029    You can #define STBIW_ZLIB_COMPRESS to use a custom zlib-style compress function
00030    for PNG compression (instead of the builtin one), it must have the following signature:
00031    unsigned char * my_compress(unsigned char *data, int data_len, int *out_len, int quality);
00032    The returned data will be freed with STBIW_FREE() (free() by default),
00033    so it must be heap allocated with STBIW_MALLOC() (malloc() by default),
00034
00035 UNICODE:
00036
00037    If compiling for Windows and you wish to use Unicode filenames, compile
00038    with
00039        #define STBIW_WINDOWS_UTF8
00040    and pass utf8-encoded filenames. Call stbiw_convert_wchar_to_utf8 to convert
00041    Windows wchar_t filenames to utf8.
00042
00043 USAGE:
00044
00045    There are five functions, one for each image file format:
00046
00047      int stbi_write_png(char const *filename, int w, int h, int comp, const void *data, int
      stride_in_bytes);
00048      int stbi_write_bmp(char const *filename, int w, int h, int comp, const void *data);
00049      int stbi_write_tga(char const *filename, int w, int h, int comp, const void *data);
00050      int stbi_write_jpg(char const *filename, int w, int h, int comp, const void *data, int quality);
00051      int stbi_write_hdr(char const *filename, int w, int h, int comp, const float *data);
00052
00053      void stbi_flip_vertically_on_write(int flag); // flag is non-zero to flip data vertically
00054
00055    There are also five equivalent functions that use an arbitrary write function. You are
00056    expected to open/close your file-equivalent before and after calling these:
00057
00058      int stbi_write_png_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const
      void  *data, int stride_in_bytes);
00059      int stbi_write_bmp_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const
      void  *data);
00060      int stbi_write_tga_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const
      void  *data);
00061      int stbi_write_hdr_to_func(stbi_write_func *func, void *context, int w, int h, int comp, const
      float *data);
00062      int stbi_write_jpg_to_func(stbi_write_func *func, void *context, int x, int y, int comp, const
      void *data, int quality);
00063
00064    where the callback is:
00065       void stbi_write_func(void *context, void *data, int size);
00066
00067    You can configure it with these global variables:
00068       int stbi_write_tga_with_rle;             // defaults to true; set to 0 to disable RLE
00069       int stbi_write_png_compression_level;    // defaults to 8; set to higher for more compression
00070       int stbi_write_force_png_filter;         // defaults to -1; set to 0..5 to force a filter mode
00071
00072
00073    You can define STBI_WRITE_NO_STDIO to disable the file variant of these
00074    functions, so the library will not use stdio.h at all. However, this will
00075    also disable HDR writing, because it requires stdio for formatted output.
00076
```

```
00077    Each function returns 0 on failure and non-0 on success.
00078
00079    The functions create an image file defined by the parameters. The image
00080    is a rectangle of pixels stored from left-to-right, top-to-bottom.
00081    Each pixel contains 'comp' channels of data stored interleaved with 8-bits
00082    per channel, in the following order: 1=Y, 2=YA, 3=RGB, 4=RGBA. (Y is
00083    monochrome color.) The rectangle is 'w' pixels wide and 'h' pixels tall.
00084    The *data pointer points to the first byte of the top-left-most pixel.
00085    For PNG, "stride_in_bytes" is the distance in bytes from the first byte of
00086    a row of pixels to the first byte of the next row of pixels.
00087
00088    PNG creates output files with the same number of components as the input.
00089    The BMP format expands Y to RGB in the file format and does not
00090    output alpha.
00091
00092    PNG supports writing rectangles of data even when the bytes storing rows of
00093    data are not consecutive in memory (e.g. sub-rectangles of a larger image),
00094    by supplying the stride between the beginning of adjacent rows. The other
00095    formats do not. (Thus you cannot write a native-format BMP through the BMP
00096    writer, both because it is in BGR order and because it may have padding
00097    at the end of the line.)
00098
00099    PNG allows you to set the deflate compression level by setting the global
00100    variable 'stbi_write_png_compression_level' (it defaults to 8).
00101
00102    HDR expects linear float data. Since the format is always 32-bit rgb(e)
00103    data, alpha (if provided) is discarded, and for monochrome data it is
00104    replicated across all three channels.
00105
00106    TGA supports RLE or non-RLE compressed data. To use non-RLE-compressed
00107    data, set the global variable 'stbi_write_tga_with_rle' to 0.
00108
00109    JPEG does ignore alpha channels in input data; quality is between 1 and 100.
00110    Higher quality looks better but results in a bigger image.
00111    JPEG baseline (no JPEG progressive).
00112
00113 CREDITS:
00114
00115
00116    Sean Barrett           -    PNG/BMP/TGA
00117    Baldur Karlsson        -    HDR
00118    Jean-Sebastien Guay    -    TGA monochrome
00119    Tim Kelsey             -    misc enhancements
00120    Alan Hickman           -    TGA RLE
00121    Emmanuel Julien        -    initial file IO callback implementation
00122    Jon Olick              -    original jo_jpeg.cpp code
00123    Daniel Gibson          -    integrate JPEG, allow external zlib
00124    Aarni Koskela          -    allow choosing PNG filter
00125
00126    bugfixes:
00127       github:Chribba
00128       Guillaume Chereau
00129       github:jry2
00130       github:romigrou
00131       Sergio Gonzalez
00132       Jonas Karlsson
00133       Filip Wasil
00134       Thatcher Ulrich
00135       github:poppolopoppo
00136       Patrick Boettcher
00137       github:xeekworx
00138       Cap Petschulat
00139       Simon Rodriguez
00140       Ivan Tikhonov
00141       github:ignotion
00142       Adam Schackart
00143       Andrew Kensler
00144
00145 LICENSE
00146
00147   See end of file for license information.
00148
00149 */
00150
00151 #ifndef INCLUDE_STB_IMAGE_WRITE_H
00152 #define INCLUDE_STB_IMAGE_WRITE_H
00153
00154 #include <stdlib.h>
00155
00156 // if STB_IMAGE_WRITE_STATIC causes problems, try defining STBIWDEF to 'inline' or 'static inline'
00157 #ifndef STBIWDEF
00158 #ifdef STB_IMAGE_WRITE_STATIC
00159 #define STBIWDEF  static
00160 #else
00161 #ifdef __cplusplus
00162 #define STBIWDEF  extern "C"
00163 #else
```

```
00164 #define STBIWDEF  extern
00165 #endif
00166 #endif
00167 #endif
00168
00169 #ifndef STB_IMAGE_WRITE_STATIC  // C++ forbids static forward declarations
00170 STBIWDEF int stbi_write_tga_with_rle;
00171 STBIWDEF int stbi_write_png_compression_level;
00172 STBIWDEF int stbi_write_force_png_filter;
00173 #endif
00174
00175 #ifndef STBI_WRITE_NO_STDIO
00176 STBIWDEF int stbi_write_png(char const *filename, int w, int h, int comp, const void  *data, int
      stride_in_bytes);
00177 STBIWDEF int stbi_write_bmp(char const *filename, int w, int h, int comp, const void  *data);
00178 STBIWDEF int stbi_write_tga(char const *filename, int w, int h, int comp, const void  *data);
00179 STBIWDEF int stbi_write_hdr(char const *filename, int w, int h, int comp, const float *data);
00180 STBIWDEF int stbi_write_jpg(char const *filename, int x, int y, int comp, const void  *data, int
      quality);
00181
00182 #ifdef STBIW_WINDOWS_UTF8
00183 STBIWDEF int stbiw_convert_wchar_to_utf8(char *buffer, size_t bufferlen, const wchar_t* input);
00184 #endif
00185 #endif
00186
00187 typedef void stbi_write_func(void *context, void *data, int size);
00188
00189 STBIWDEF int stbi_write_png_to_func(stbi_write_func *func, void *context, int w, int h, int comp,
      const void  *data, int stride_in_bytes);
00190 STBIWDEF int stbi_write_bmp_to_func(stbi_write_func *func, void *context, int w, int h, int comp,
      const void  *data);
00191 STBIWDEF int stbi_write_tga_to_func(stbi_write_func *func, void *context, int w, int h, int comp,
      const void  *data);
00192 STBIWDEF int stbi_write_hdr_to_func(stbi_write_func *func, void *context, int w, int h, int comp,
      const float *data);
00193 STBIWDEF int stbi_write_jpg_to_func(stbi_write_func *func, void *context, int x, int y, int comp,
      const void  *data, int quality);
00194
00195 STBIWDEF void stbi_flip_vertically_on_write(int flip_boolean);
00196
00197 #endif//INCLUDE_STB_IMAGE_WRITE_H
00198
00199 #ifdef STB_IMAGE_WRITE_IMPLEMENTATION
00200
00201 #ifdef _WIN32
00202    #ifndef _CRT_SECURE_NO_WARNINGS
00203    #define _CRT_SECURE_NO_WARNINGS
00204    #endif
00205    #ifndef _CRT_NONSTDC_NO_DEPRECATE
00206    #define _CRT_NONSTDC_NO_DEPRECATE
00207    #endif
00208 #endif
00209
00210 #ifndef STBI_WRITE_NO_STDIO
00211 #include <stdio.h>
00212 #endif // STBI_WRITE_NO_STDIO
00213
00214 #include <stdarg.h>
00215 #include <stdlib.h>
00216 #include <string.h>
00217 #include <math.h>
00218
00219 #if defined(STBIW_MALLOC) && defined(STBIW_FREE) && (defined(STBIW_REALLOC) ||
      defined(STBIW_REALLOC_SIZED))
00220 // ok
00221 #elif !defined(STBIW_MALLOC) && !defined(STBIW_FREE) && !defined(STBIW_REALLOC) &&
      !defined(STBIW_REALLOC_SIZED)
00222 // ok
00223 #else
00224 #error "Must define all or none of STBIW_MALLOC, STBIW_FREE, and STBIW_REALLOC (or
      STBIW_REALLOC_SIZED)."
00225 #endif
00226
00227 #ifndef STBIW_MALLOC
00228 #define STBIW_MALLOC(sz)        malloc(sz)
00229 #define STBIW_REALLOC(p,newsz)  realloc(p,newsz)
00230 #define STBIW_FREE(p)           free(p)
00231 #endif
00232
00233 #ifndef STBIW_REALLOC_SIZED
00234 #define STBIW_REALLOC_SIZED(p,oldsz,newsz) STBIW_REALLOC(p,newsz)
00235 #endif
00236
00237
00238 #ifndef STBIW_MEMMOVE
00239 #define STBIW_MEMMOVE(a,b,sz) memmove(a,b,sz)
00240 #endif
```

```
00241
00242
00243 #ifndef STBIW_ASSERT
00244 #include <assert.h>
00245 #define STBIW_ASSERT(x) assert(x)
00246 #endif
00247
00248 #define STBIW_UCHAR(x) (unsigned char) ((x) & 0xff)
00249
00250 #ifdef STB_IMAGE_WRITE_STATIC
00251 static int stbi_write_png_compression_level = 8;
00252 static int stbi_write_tga_with_rle = 1;
00253 static int stbi_write_force_png_filter = -1;
00254 #else
00255 int stbi_write_png_compression_level = 8;
00256 int stbi_write_tga_with_rle = 1;
00257 int stbi_write_force_png_filter = -1;
00258 #endif
00259
00260 static int stbi__flip_vertically_on_write = 0;
00261
00262 STBIWDEF void stbi_flip_vertically_on_write(int flag)
00263 {
00264    stbi__flip_vertically_on_write = flag;
00265 }
00266
00267 typedef struct
00268 {
00269    stbi_write_func *func;
00270    void *context;
00271    unsigned char buffer[64];
00272    int buf_used;
00273 } stbi__write_context;
00274
00275 // initialize a callback-based context
00276 static void stbi__start_write_callbacks(stbi__write_context *s, stbi_write_func *c, void *context)
00277 {
00278    s->func    = c;
00279    s->context = context;
00280 }
00281
00282 #ifndef STBI_WRITE_NO_STDIO
00283
00284 static void stbi__stdio_write(void *context, void *data, int size)
00285 {
00286    fwrite(data,1,size,(FILE*) context);
00287 }
00288
00289 #if defined(_WIN32) && defined(STBIW_WINDOWS_UTF8)
00290 #ifdef __cplusplus
00291 #define STBIW_EXTERN extern "C"
00292 #else
00293 #define STBIW_EXTERN extern
00294 #endif
00295 STBIW_EXTERN __declspec(dllimport) int __stdcall MultiByteToWideChar(unsigned int cp, unsigned long
      flags, const char *str, int cbmb, wchar_t *widestr, int cchwide);
00296 STBIW_EXTERN __declspec(dllimport) int __stdcall WideCharToMultiByte(unsigned int cp, unsigned long
      flags, const wchar_t *widestr, int cchwide, char *str, int cbmb, const char *defchar, int
      *used_default);
00297
00298 STBIWDEF int stbiw_convert_wchar_to_utf8(char *buffer, size_t bufferlen, const wchar_t* input)
00299 {
00300    return WideCharToMultiByte(65001 /* UTF8 */, 0, input, -1, buffer, (int) bufferlen, NULL, NULL);
00301 }
00302 #endif
00303
00304 static FILE *stbiw__fopen(char const *filename, char const *mode)
00305 {
00306    FILE *f;
00307 #if defined(_WIN32) && defined(STBIW_WINDOWS_UTF8)
00308    wchar_t wMode[64];
00309    wchar_t wFilename[1024];
00310    if (0 == MultiByteToWideChar(65001 /* UTF8 */, 0, filename, -1, wFilename,
      sizeof(wFilename)/sizeof(*wFilename)))
00311       return 0;
00312
00313    if (0 == MultiByteToWideChar(65001 /* UTF8 */, 0, mode, -1, wMode, sizeof(wMode)/sizeof(*wMode)))
00314       return 0;
00315
00316 #if defined(_MSC_VER) && _MSC_VER >= 1400
00317    if (0 != _wfopen_s(&f, wFilename, wMode))
00318       f = 0;
00319 #else
00320    f = _wfopen(wFilename, wMode);
00321 #endif
00322
00323 #elif defined(_MSC_VER) && _MSC_VER >= 1400
```

```
00324    if (0 != fopen_s(&f, filename, mode))
00325       f=0;
00326 #else
00327    f = fopen(filename, mode);
00328 #endif
00329    return f;
00330 }
00331
00332 static int stbi__start_write_file(stbi__write_context *s, const char *filename)
00333 {
00334    FILE *f = stbiw__fopen(filename, "wb");
00335    stbi__start_write_callbacks(s, stbi__stdio_write, (void *) f);
00336    return f != NULL;
00337 }
00338
00339 static void stbi__end_write_file(stbi__write_context *s)
00340 {
00341    fclose((FILE *)s->context);
00342 }
00343
00344 #endif // !STBI_WRITE_NO_STDIO
00345
00346 typedef unsigned int stbiw_uint32;
00347 typedef int stb_image_write_test[sizeof(stbiw_uint32)==4 ? 1 : -1];
00348
00349 static void stbiw__writefv(stbi__write_context *s, const char *fmt, va_list v)
00350 {
00351    while (*fmt) {
00352       switch (*fmt++) {
00353          case ' ': break;
00354          case '1': { unsigned char x = STBIW_UCHAR(va_arg(v, int));
00355                      s->func(s->context,&x,1);
00356                      break; }
00357          case '2': { int x = va_arg(v,int);
00358                      unsigned char b[2];
00359                      b[0] = STBIW_UCHAR(x);
00360                      b[1] = STBIW_UCHAR(x>>8);
00361                      s->func(s->context,b,2);
00362                      break; }
00363          case '4': { stbiw_uint32 x = va_arg(v,int);
00364                      unsigned char b[4];
00365                      b[0]=STBIW_UCHAR(x);
00366                      b[1]=STBIW_UCHAR(x>>8);
00367                      b[2]=STBIW_UCHAR(x>>16);
00368                      b[3]=STBIW_UCHAR(x>>24);
00369                      s->func(s->context,b,4);
00370                      break; }
00371          default:
00372             STBIW_ASSERT(0);
00373             return;
00374       }
00375    }
00376 }
00377
00378 static void stbiw__writef(stbi__write_context *s, const char *fmt, ...)
00379 {
00380    va_list v;
00381    va_start(v, fmt);
00382    stbiw__writefv(s, fmt, v);
00383    va_end(v);
00384 }
00385
00386 static void stbiw__write_flush(stbi__write_context *s)
00387 {
00388    if (s->buf_used) {
00389       s->func(s->context, &s->buffer, s->buf_used);
00390       s->buf_used = 0;
00391    }
00392 }
00393
00394 static void stbiw__putc(stbi__write_context *s, unsigned char c)
00395 {
00396    s->func(s->context, &c, 1);
00397 }
00398
00399 static void stbiw__write1(stbi__write_context *s, unsigned char a)
00400 {
00401    if ((size_t)s->buf_used + 1 > sizeof(s->buffer))
00402       stbiw__write_flush(s);
00403    s->buffer[s->buf_used++] = a;
00404 }
00405
00406 static void stbiw__write3(stbi__write_context *s, unsigned char a, unsigned char b, unsigned char c)
00407 {
00408    int n;
00409    if ((size_t)s->buf_used + 3 > sizeof(s->buffer))
00410       stbiw__write_flush(s);
```

```
00411    n = s->buf_used;
00412    s->buf_used = n+3;
00413    s->buffer[n+0] = a;
00414    s->buffer[n+1] = b;
00415    s->buffer[n+2] = c;
00416 }
00417
00418 static void stbiw__write_pixel(stbi__write_context *s, int rgb_dir, int comp, int write_alpha, int
       expand_mono, unsigned char *d)
00419 {
00420    unsigned char bg[3] = { 255, 0, 255}, px[3];
00421    int k;
00422
00423    if (write_alpha < 0)
00424       stbiw__write1(s, d[comp - 1]);
00425
00426    switch (comp) {
00427       case 2: // 2 pixels = mono + alpha, alpha is written separately, so same as 1-channel case
00428       case 1:
00429          if (expand_mono)
00430             stbiw__write3(s, d[0], d[0], d[0]); // monochrome bmp
00431          else
00432             stbiw__write1(s, d[0]);  // monochrome TGA
00433          break;
00434       case 4:
00435          if (!write_alpha) {
00436             // composite against pink background
00437             for (k = 0; k < 3; ++k)
00438                px[k] = bg[k] + ((d[k] - bg[k]) * d[3]) / 255;
00439             stbiw__write3(s, px[1 - rgb_dir], px[1], px[1 + rgb_dir]);
00440             break;
00441          }
00442          /* FALLTHROUGH */
00443       case 3:
00444          stbiw__write3(s, d[1 - rgb_dir], d[1], d[1 + rgb_dir]);
00445          break;
00446    }
00447    if (write_alpha > 0)
00448       stbiw__write1(s, d[comp - 1]);
00449 }
00450
00451 static void stbiw__write_pixels(stbi__write_context *s, int rgb_dir, int vdir, int x, int y, int comp,
       void *data, int write_alpha, int scanline_pad, int expand_mono)
00452 {
00453    stbiw_uint32 zero = 0;
00454    int i,j, j_end;
00455
00456    if (y <= 0)
00457       return;
00458
00459    if (stbi__flip_vertically_on_write)
00460       vdir *= -1;
00461
00462    if (vdir < 0) {
00463       j_end = -1; j = y-1;
00464    } else {
00465       j_end =  y; j = 0;
00466    }
00467
00468    for (; j != j_end; j += vdir) {
00469       for (i=0; i < x; ++i) {
00470          unsigned char *d = (unsigned char *) data + (j*x+i)*comp;
00471          stbiw__write_pixel(s, rgb_dir, comp, write_alpha, expand_mono, d);
00472       }
00473       stbiw__write_flush(s);
00474       s->func(s->context, &zero, scanline_pad);
00475    }
00476 }
00477
00478 static int stbiw__outfile(stbi__write_context *s, int rgb_dir, int vdir, int x, int y, int comp, int
       expand_mono, void *data, int alpha, int pad, const char *fmt, ...)
00479 {
00480    if (y < 0 || x < 0) {
00481       return 0;
00482    } else {
00483       va_list v;
00484       va_start(v, fmt);
00485       stbiw__writefv(s, fmt, v);
00486       va_end(v);
00487       stbiw__write_pixels(s,rgb_dir,vdir,x,y,comp,data,alpha,pad, expand_mono);
00488       return 1;
00489    }
00490 }
00491
00492 static int stbi_write_bmp_core(stbi__write_context *s, int x, int y, int comp, const void *data)
00493 {
00494    if (comp != 4) {
```

```
00495       // write RGB bitmap
00496       int pad = (-x*3) & 3;
00497       return stbiw__outfile(s,-1,-1,x,y,comp,1,(void *) data,0,pad,
00498               "11 4 22 4" "4 44 22 444444",
00499               'B', 'M', 14+40+(x*3+pad)*y, 0,0, 14+40,  // file header
00500                40, x,y, 1,24, 0,0,0,0,0,0);             // bitmap header
00501    } else {
00502       // RGBA bitmaps need a v4 header
00503       // use BI_BITFIELDS mode with 32bpp and alpha mask
00504       // (straight BI_RGB with alpha mask doesn't work in most readers)
00505       return stbiw__outfile(s,-1,-1,x,y,comp,1,(void *)data,1,0,
00506          "11 4 22 4" "4 44 22 444444 4444 4 444 444 444 444",
00507          'B', 'M', 14+108+x*y*4, 0, 0, 14+108, // file header
00508          108, x,y, 1,32, 3,0,0,0,0,0, 0xff0000,0xff00,0xff,0xff000000u, 0, 0,0,0, 0,0,0, 0,0,0,
      0,0,0); // bitmap V4 header
00509    }
00510 }
00511
00512 STBIWDEF int stbi_write_bmp_to_func(stbi_write_func *func, void *context, int x, int y, int comp,
      const void *data)
00513 {
00514    stbi__write_context s = { 0 };
00515    stbi__start_write_callbacks(&s, func, context);
00516    return stbi_write_bmp_core(&s, x, y, comp, data);
00517 }
00518
00519 #ifndef STBI_WRITE_NO_STDIO
00520 STBIWDEF int stbi_write_bmp(char const *filename, int x, int y, int comp, const void *data)
00521 {
00522    stbi__write_context s = { 0 };
00523    if (stbi__start_write_file(&s,filename)) {
00524       int r = stbi_write_bmp_core(&s, x, y, comp, data);
00525       stbi__end_write_file(&s);
00526       return r;
00527    } else
00528       return 0;
00529 }
00530 #endif
00531
00532 static int stbi_write_tga_core(stbi__write_context *s, int x, int y, int comp, void *data)
00533 {
00534    int has_alpha = (comp == 2 || comp == 4);
00535    int colorbytes = has_alpha ? comp-1 : comp;
00536    int format = colorbytes < 2 ? 3 : 2; // 3 color channels (RGB/RGBA) = 2, 1 color channel (Y/YA) = 3
00537
00538    if (y < 0 || x < 0)
00539       return 0;
00540
00541    if (!stbi_write_tga_with_rle) {
00542       return stbiw__outfile(s, -1, -1, x, y, comp, 0, (void *) data, has_alpha, 0,
00543          "111 221 2222 11", 0, 0, format, 0, 0, 0, 0, 0, x, y, (colorbytes + has_alpha) * 8, has_alpha
      * 8);
00544    } else {
00545       int i,j,k;
00546       int jend, jdir;
00547
00548       stbiw__writef(s, "111 221 2222 11", 0,0,format+8, 0,0,0, 0,0,x,y, (colorbytes + has_alpha) * 8,
      has_alpha * 8);
00549
00550       if (stbi__flip_vertically_on_write) {
00551          j = 0;
00552          jend = y;
00553          jdir = 1;
00554       } else {
00555          j = y-1;
00556          jend = -1;
00557          jdir = -1;
00558       }
00559       for (; j != jend; j += jdir) {
00560          unsigned char *row = (unsigned char *) data + j * x * comp;
00561          int len;
00562
00563          for (i = 0; i < x; i += len) {
00564             unsigned char *begin = row + i * comp;
00565             int diff = 1;
00566             len = 1;
00567
00568             if (i < x - 1) {
00569                ++len;
00570                diff = memcmp(begin, row + (i + 1) * comp, comp);
00571                if (diff) {
00572                   const unsigned char *prev = begin;
00573                   for (k = i + 2; k < x && len < 128; ++k) {
00574                      if (memcmp(prev, row + k * comp, comp)) {
00575                         prev += comp;
00576                         ++len;
00577                      } else {
```

```
00578                         --len;
00579                         break;
00580                      }
00581                   }
00582                } else {
00583                   for (k = i + 2; k < x && len < 128; ++k) {
00584                      if (!memcmp(begin, row + k * comp, comp)) {
00585                         ++len;
00586                      } else {
00587                         break;
00588                      }
00589                   }
00590                }
00591             }
00592
00593             if (diff) {
00594                unsigned char header = STBIW_UCHAR(len - 1);
00595                stbiw__write1(s, header);
00596                for (k = 0; k < len; ++k) {
00597                   stbiw__write_pixel(s, -1, comp, has_alpha, 0, begin + k * comp);
00598                }
00599             } else {
00600                unsigned char header = STBIW_UCHAR(len - 129);
00601                stbiw__write1(s, header);
00602                stbiw__write_pixel(s, -1, comp, has_alpha, 0, begin);
00603             }
00604          }
00605       }
00606       stbiw__write_flush(s);
00607    }
00608    return 1;
00609 }
00610
00611 STBIWDEF int stbi_write_tga_to_func(stbi_write_func *func, void *context, int x, int y, int comp,
      const void *data)
00612 {
00613    stbi__write_context s = { 0 };
00614    stbi__start_write_callbacks(&s, func, context);
00615    return stbi_write_tga_core(&s, x, y, comp, (void *) data);
00616 }
00617
00618 #ifndef STBI_WRITE_NO_STDIO
00619 STBIWDEF int stbi_write_tga(char const *filename, int x, int y, int comp, const void *data)
00620 {
00621    stbi__write_context s = { 0 };
00622    if (stbi__start_write_file(&s,filename)) {
00623       int r = stbi_write_tga_core(&s, x, y, comp, (void *) data);
00624       stbi__end_write_file(&s);
00625       return r;
00626    } else
00627       return 0;
00628 }
00629 #endif
00630
00631 // *************************************************************************************************
00632 // Radiance RGBE HDR writer
00633 // by Baldur Karlsson
00634
00635 #define stbiw__max(a, b)  ((a) > (b) ? (a) : (b))
00636
00637 #ifndef STBI_WRITE_NO_STDIO
00638
00639 static void stbiw__linear_to_rgbe(unsigned char *rgbe, float *linear)
00640 {
00641    int exponent;
00642    float maxcomp = stbiw__max(linear[0], stbiw__max(linear[1], linear[2]));
00643
00644    if (maxcomp < 1e-32f) {
00645       rgbe[0] = rgbe[1] = rgbe[2] = rgbe[3] = 0;
00646    } else {
00647       float normalize = (float) frexp(maxcomp, &exponent) * 256.0f/maxcomp;
00648
00649       rgbe[0] = (unsigned char)(linear[0] * normalize);
00650       rgbe[1] = (unsigned char)(linear[1] * normalize);
00651       rgbe[2] = (unsigned char)(linear[2] * normalize);
00652       rgbe[3] = (unsigned char)(exponent + 128);
00653    }
00654 }
00655
00656 static void stbiw__write_run_data(stbi__write_context *s, int length, unsigned char databyte)
00657 {
00658    unsigned char lengthbyte = STBIW_UCHAR(length+128);
00659    STBIW_ASSERT(length+128 <= 255);
00660    s->func(s->context, &lengthbyte, 1);
00661    s->func(s->context, &databyte, 1);
00662 }
00663
```

```
00664 static void stbiw__write_dump_data(stbi__write_context *s, int length, unsigned char *data)
00665 {
00666    unsigned char lengthbyte = STBIW_UCHAR(length);
00667    STBIW_ASSERT(length <= 128); // inconsistent with spec but consistent with official code
00668    s->func(s->context, &lengthbyte, 1);
00669    s->func(s->context, data, length);
00670 }
00671
00672 static void stbiw__write_hdr_scanline(stbi__write_context *s, int width, int ncomp, unsigned char
      *scratch, float *scanline)
00673 {
00674    unsigned char scanlineheader[4] = { 2, 2, 0, 0 };
00675    unsigned char rgbe[4];
00676    float linear[3];
00677    int x;
00678
00679    scanlineheader[2] = (width&0xff00)»8;
00680    scanlineheader[3] = (width&0x00ff);
00681
00682    /* skip RLE for images too small or large */
00683    if (width < 8 || width >= 32768) {
00684       for (x=0; x < width; x++) {
00685          switch (ncomp) {
00686             case 4: /* fallthrough */
00687             case 3: linear[2] = scanline[x*ncomp + 2];
00688                     linear[1] = scanline[x*ncomp + 1];
00689                     linear[0] = scanline[x*ncomp + 0];
00690                     break;
00691             default:
00692                     linear[0] = linear[1] = linear[2] = scanline[x*ncomp + 0];
00693                     break;
00694          }
00695          stbiw__linear_to_rgbe(rgbe, linear);
00696          s->func(s->context, rgbe, 4);
00697       }
00698    } else {
00699       int c,r;
00700       /* encode into scratch buffer */
00701       for (x=0; x < width; x++) {
00702          switch(ncomp) {
00703             case 4: /* fallthrough */
00704             case 3: linear[2] = scanline[x*ncomp + 2];
00705                     linear[1] = scanline[x*ncomp + 1];
00706                     linear[0] = scanline[x*ncomp + 0];
00707                     break;
00708             default:
00709                     linear[0] = linear[1] = linear[2] = scanline[x*ncomp + 0];
00710                     break;
00711          }
00712          stbiw__linear_to_rgbe(rgbe, linear);
00713          scratch[x + width*0] = rgbe[0];
00714          scratch[x + width*1] = rgbe[1];
00715          scratch[x + width*2] = rgbe[2];
00716          scratch[x + width*3] = rgbe[3];
00717       }
00718
00719       s->func(s->context, scanlineheader, 4);
00720
00721       /* RLE each component separately */
00722       for (c=0; c < 4; c++) {
00723          unsigned char *comp = &scratch[width*c];
00724
00725          x = 0;
00726          while (x < width) {
00727             // find first run
00728             r = x;
00729             while (r+2 < width) {
00730                if (comp[r] == comp[r+1] && comp[r] == comp[r+2])
00731                   break;
00732                ++r;
00733             }
00734             if (r+2 >= width)
00735                r = width;
00736             // dump up to first run
00737             while (x < r) {
00738                int len = r-x;
00739                if (len > 128) len = 128;
00740                stbiw__write_dump_data(s, len, &comp[x]);
00741                x += len;
00742             }
00743             // if there's a run, output it
00744             if (r+2 < width) { // same test as what we break out of in search loop, so only true if we
      break'd
00745                // find next byte after run
00746                while (r < width && comp[r] == comp[x])
00747                   ++r;
00748                // output run up to r
```

```
00749                while (x < r) {
00750                   int len = r-x;
00751                   if (len > 127) len = 127;
00752                   stbiw__write_run_data(s, len, comp[x]);
00753                   x += len;
00754                }
00755             }
00756          }
00757       }
00758    }
00759 }
00760
00761 static int stbi_write_hdr_core(stbi__write_context *s, int x, int y, int comp, float *data)
00762 {
00763    if (y <= 0 || x <= 0 || data == NULL)
00764       return 0;
00765    else {
00766       // Each component is stored separately. Allocate scratch space for full output scanline.
00767       unsigned char *scratch = (unsigned char *) STBIW_MALLOC(x*4);
00768       int i, len;
00769       char buffer[128];
00770       char header[] = "#?RADIANCE\n# Written by stb_image_write.h\nFORMAT=32-bit_rle_rgbe\n";
00771       s->func(s->context, header, sizeof(header)-1);
00772
00773 #ifdef __STDC_LIB_EXT1__
00774       len = sprintf_s(buffer, sizeof(buffer), "EXPOSURE=          1.0000000000000\n\n-Y %d +X %d\n",
      y, x);
00775 #else
00776       len = sprintf(buffer, "EXPOSURE=          1.0000000000000\n\n-Y %d +X %d\n", y, x);
00777 #endif
00778       s->func(s->context, buffer, len);
00779
00780       for(i=0; i < y; i++)
00781          stbiw__write_hdr_scanline(s, x, comp, scratch, data + comp*x*(stbi__flip_vertically_on_write
      ? y-1-i : i));
00782       STBIW_FREE(scratch);
00783       return 1;
00784    }
00785 }
00786
00787 STBIWDEF int stbi_write_hdr_to_func(stbi_write_func *func, void *context, int x, int y, int comp,
      const float *data)
00788 {
00789    stbi__write_context s = { 0 };
00790    stbi__start_write_callbacks(&s, func, context);
00791    return stbi_write_hdr_core(&s, x, y, comp, (float *) data);
00792 }
00793
00794 STBIWDEF int stbi_write_hdr(char const *filename, int x, int y, int comp, const float *data)
00795 {
00796    stbi__write_context s = { 0 };
00797    if (stbi__start_write_file(&s,filename)) {
00798       int r = stbi_write_hdr_core(&s, x, y, comp, (float *) data);
00799       stbi__end_write_file(&s);
00800       return r;
00801    } else
00802       return 0;
00803 }
00804 #endif // STBI_WRITE_NO_STDIO
00805
00806
00808 //
00809 // PNG writer
00810 //
00811
00812 #ifndef STBIW_ZLIB_COMPRESS
00813 // stretchy buffer; stbiw__sbpush() == vector<>::push_back() -- stbiw__sbcount() == vector<>::size()
00814 #define stbiw__sbraw(a) ((int *) (void *) (a) - 2)
00815 #define stbiw__sbm(a)   stbiw__sbraw(a)[0]
00816 #define stbiw__sbn(a)   stbiw__sbraw(a)[1]
00817
00818 #define stbiw__sbneedgrow(a,n)  ((a)==0 || stbiw__sbn(a)+n >= stbiw__sbm(a))
00819 #define stbiw__sbmaybegrow(a,n) (stbiw__sbneedgrow(a,(n)) ? stbiw__sbgrow(a,n) : 0)
00820 #define stbiw__sbgrow(a,n)  stbiw__sbgrowf((void **) &(a), (n), sizeof(*(a)))
00821
00822 #define stbiw__sbpush(a, v)      (stbiw__sbmaybegrow(a,1), (a)[stbiw__sbn(a)++] = (v))
00823 #define stbiw__sbcount(a)        ((a) ? stbiw__sbn(a) : 0)
00824 #define stbiw__sbfree(a)         ((a) ? STBIW_FREE(stbiw__sbraw(a)),0 : 0)
00825
00826 static void *stbiw__sbgrowf(void **arr, int increment, int itemsize)
00827 {
00828    int m = *arr ? 2*stbiw__sbm(*arr)+increment : increment+1;
00829    void *p = STBIW_REALLOC_SIZED(*arr ? stbiw__sbraw(*arr) : 0, *arr ? (stbiw__sbm(*arr)*itemsize +
      sizeof(int)*2) : 0, itemsize * m + sizeof(int)*2);
00830    STBIW_ASSERT(p);
00831    if (p) {
00832       if (!*arr) ((int *) p)[1] = 0;
```

```
00833       *arr = (void *) ((int *) p + 2);
00834       stbiw__sbm(*arr) = m;
00835    }
00836    return *arr;
00837 }
00838
00839 static unsigned char *stbiw__zlib_flushf(unsigned char *data, unsigned int *bitbuffer, int *bitcount)
00840 {
00841    while (*bitcount >= 8) {
00842       stbiw__sbpush(data, STBIW_UCHAR(*bitbuffer));
00843       *bitbuffer »= 8;
00844       *bitcount -= 8;
00845    }
00846    return data;
00847 }
00848
00849 static int stbiw__zlib_bitrev(int code, int codebits)
00850 {
00851    int res=0;
00852    while (codebits--) {
00853       res = (res « 1) | (code & 1);
00854       code »= 1;
00855    }
00856    return res;
00857 }
00858
00859 static unsigned int stbiw__zlib_countm(unsigned char *a, unsigned char *b, int limit)
00860 {
00861    int i;
00862    for (i=0; i < limit && i < 258; ++i)
00863       if (a[i] != b[i]) break;
00864    return i;
00865 }
00866
00867 static unsigned int stbiw__zhash(unsigned char *data)
00868 {
00869    stbiw_uint32 hash = data[0] + (data[1] « 8) + (data[2] « 16);
00870    hash ^= hash « 3;
00871    hash += hash » 5;
00872    hash ^= hash « 4;
00873    hash += hash » 17;
00874    hash ^= hash « 25;
00875    hash += hash » 6;
00876    return hash;
00877 }
00878
00879 #define stbiw__zlib_flush() (out = stbiw__zlib_flushf(out, &bitbuf, &bitcount))
00880 #define stbiw__zlib_add(code,codebits) \
00881       (bitbuf |= (code) « bitcount, bitcount += (codebits), stbiw__zlib_flush())
00882 #define stbiw__zlib_huffa(b,c)  stbiw__zlib_add(stbiw__zlib_bitrev(b,c),c)
00883 // default huffman tables
00884 #define stbiw__zlib_huff1(n)  stbiw__zlib_huffa(0x30 + (n), 8)
00885 #define stbiw__zlib_huff2(n)  stbiw__zlib_huffa(0x190 + (n)-144, 9)
00886 #define stbiw__zlib_huff3(n)  stbiw__zlib_huffa(0 + (n)-256,7)
00887 #define stbiw__zlib_huff4(n)  stbiw__zlib_huffa(0xc0 + (n)-280,8)
00888 #define stbiw__zlib_huff(n)  ((n) <= 143 ? stbiw__zlib_huff1(n) : (n) <= 255 ? stbiw__zlib_huff2(n) :
      (n) <= 279 ? stbiw__zlib_huff3(n) : stbiw__zlib_huff4(n))
00889 #define stbiw__zlib_huffb(n) ((n) <= 143 ? stbiw__zlib_huff1(n) : stbiw__zlib_huff2(n))
00890
00891 #define stbiw__ZHASH   16384
00892
00893 #endif // STBIW_ZLIB_COMPRESS
00894
00895 STBIWDEF unsigned char * stbi_zlib_compress(unsigned char *data, int data_len, int *out_len, int
      quality)
00896 {
00897 #ifdef STBIW_ZLIB_COMPRESS
00898    // user provided a zlib compress implementation, use that
00899    return STBIW_ZLIB_COMPRESS(data, data_len, out_len, quality);
00900 #else // use builtin
00901    static unsigned short lengthc[] = {
      3,4,5,6,7,8,9,10,11,13,15,17,19,23,27,31,35,43,51,59,67,83,99,115,131,163,195,227,258, 259 };
00902    static unsigned char  lengtheb[]= { 0,0,0,0,0,0,0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4,
      4,  5,  5,  5,  5,  0 };
00903    static unsigned short distc[]   = {
      1,2,3,4,5,7,9,13,17,25,33,49,65,97,129,193,257,385,513,769,1025,1537,2049,3073,4097,6145,8193,12289,16385,24577,
      32768 };
00904    static unsigned char  disteb[]  = {
      0,0,0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,12,13,13 };
00905    unsigned int bitbuf=0;
00906    int i,j, bitcount=0;
00907    unsigned char *out = NULL;
00908    unsigned char ***hash_table = (unsigned char***) STBIW_MALLOC(stbiw__ZHASH * sizeof(unsigned
      char**));
00909    if (hash_table == NULL)
00910       return NULL;
00911    if (quality < 5) quality = 5;
```

```
00912
00913    stbiw__sbpush(out, 0x78);   // DEFLATE 32K window
00914    stbiw__sbpush(out, 0x5e);   // FLEVEL = 1
00915    stbiw__zlib_add(1,1);  // BFINAL = 1
00916    stbiw__zlib_add(1,2);  // BTYPE = 1 -- fixed huffman
00917
00918    for (i=0; i < stbiw__ZHASH; ++i)
00919       hash_table[i] = NULL;
00920
00921    i=0;
00922    while (i < data_len-3) {
00923       // hash next 3 bytes of data to be compressed
00924       int h = stbiw__zhash(data+i)&(stbiw__ZHASH-1), best=3;
00925       unsigned char *bestloc = 0;
00926       unsigned char **hlist = hash_table[h];
00927       int n = stbiw__sbcount(hlist);
00928       for (j=0; j < n; ++j) {
00929          if (hlist[j]-data > i-32768) { // if entry lies within window
00930             int d = stbiw__zlib_countm(hlist[j], data+i, data_len-i);
00931             if (d >= best) { best=d; bestloc=hlist[j]; }
00932          }
00933       }
00934       // when hash table entry is too long, delete half the entries
00935       if (hash_table[h] && stbiw__sbn(hash_table[h]) == 2*quality) {
00936          STBIW_MEMMOVE(hash_table[h], hash_table[h]+quality, sizeof(hash_table[h][0])*quality);
00937          stbiw__sbn(hash_table[h]) = quality;
00938       }
00939       stbiw__sbpush(hash_table[h],data+i);
00940
00941       if (bestloc) {
00942          // "lazy matching" - check match at *next* byte, and if it's better, do cur byte as literal
00943          h = stbiw__zhash(data+i+1)&(stbiw__ZHASH-1);
00944          hlist = hash_table[h];
00945          n = stbiw__sbcount(hlist);
00946          for (j=0; j < n; ++j) {
00947             if (hlist[j]-data > i-32767) {
00948                int e = stbiw__zlib_countm(hlist[j], data+i+1, data_len-i-1);
00949                if (e > best) { // if next match is better, bail on current match
00950                   bestloc = NULL;
00951                   break;
00952                }
00953             }
00954          }
00955       }
00956
00957       if (bestloc) {
00958          int d = (int) (data+i - bestloc); // distance back
00959          STBIW_ASSERT(d <= 32767 && best <= 258);
00960          for (j=0; best > lengthc[j+1]-1; ++j);
00961          stbiw__zlib_huff(j+257);
00962          if (lengtheb[j]) stbiw__zlib_add(best - lengthc[j], lengtheb[j]);
00963          for (j=0; d > distc[j+1]-1; ++j);
00964          stbiw__zlib_add(stbiw__zlib_bitrev(j,5),5);
00965          if (disteb[j]) stbiw__zlib_add(d - distc[j], disteb[j]);
00966          i += best;
00967       } else {
00968          stbiw__zlib_huffb(data[i]);
00969          ++i;
00970       }
00971    }
00972    // write out final bytes
00973    for (;i < data_len; ++i)
00974       stbiw__zlib_huffb(data[i]);
00975    stbiw__zlib_huff(256); // end of block
00976    // pad with 0 bits to byte boundary
00977    while (bitcount)
00978       stbiw__zlib_add(0,1);
00979
00980    for (i=0; i < stbiw__ZHASH; ++i)
00981       (void) stbiw__sbfree(hash_table[i]);
00982    STBIW_FREE(hash_table);
00983
00984    // store uncompressed instead if compression was worse
00985    if (stbiw__sbn(out) > data_len + 2 + ((data_len+32766)/32767)*5) {
00986       stbiw__sbn(out) = 2;  // truncate to DEFLATE 32K window and FLEVEL = 1
00987       for (j = 0; j < data_len;) {
00988          int blocklen = data_len - j;
00989          if (blocklen > 32767) blocklen = 32767;
00990          stbiw__sbpush(out, data_len - j == blocklen); // BFINAL = ?, BTYPE = 0 -- no compression
00991          stbiw__sbpush(out, STBIW_UCHAR(blocklen)); // LEN
00992          stbiw__sbpush(out, STBIW_UCHAR(blocklen >> 8));
00993          stbiw__sbpush(out, STBIW_UCHAR(~blocklen)); // NLEN
00994          stbiw__sbpush(out, STBIW_UCHAR(~blocklen >> 8));
00995          memcpy(out+stbiw__sbn(out), data+j, blocklen);
00996          stbiw__sbn(out) += blocklen;
00997          j += blocklen;
00998       }
```

```
00999    }
01000
01001    {
01002       // compute adler32 on input
01003       unsigned int s1=1, s2=0;
01004       int blocklen = (int) (data_len % 5552);
01005       j=0;
01006       while (j < data_len) {
01007          for (i=0; i < blocklen; ++i) { s1 += data[j+i]; s2 += s1; }
01008          s1 %= 65521; s2 %= 65521;
01009          j += blocklen;
01010          blocklen = 5552;
01011       }
01012       stbiw__sbpush(out, STBIW_UCHAR(s2 » 8));
01013       stbiw__sbpush(out, STBIW_UCHAR(s2));
01014       stbiw__sbpush(out, STBIW_UCHAR(s1 » 8));
01015       stbiw__sbpush(out, STBIW_UCHAR(s1));
01016    }
01017    *out_len = stbiw__sbn(out);
01018    // make returned pointer freeable
01019    STBIW_MEMMOVE(stbiw__sbraw(out), out, *out_len);
01020    return (unsigned char *) stbiw__sbraw(out);
01021 #endif // STBIW_ZLIB_COMPRESS
01022 }
01023
01024 static unsigned int stbiw__crc32(unsigned char *buffer, int len)
01025 {
01026 #ifdef STBIW_CRC32
01027     return STBIW_CRC32(buffer, len);
01028 #else
01029    static unsigned int crc_table[256] =
01030    {
01031       0x00000000, 0x77073096, 0xEE0E612C, 0x990951BA, 0x076DC419, 0x706AF48F, 0xE963A535, 0x9E6495A3,
01032       0x0eDB8832, 0x79DCB8A4, 0xE0D5E91E, 0x97D2D988, 0x09B64C2B, 0x7EB17CBD, 0xE7B82D07, 0x90BF1D91,
01033       0x1DB71064, 0x6AB020F2, 0xF3B97148, 0x84BE41DE, 0x1ADAD47D, 0x6DDDE4EB, 0xF4D4B551, 0x83D385C7,
01034       0x136C9856, 0x646BA8C0, 0xFD62F97A, 0x8A65C9EC, 0x14015C4F, 0x63066CD9, 0xFA0F3D63, 0x8D080DF5,
01035       0x3B6E20C8, 0x4C69105E, 0xD56041E4, 0xA2677172, 0x3C03E4D1, 0x4B04D447, 0xD20D85FD, 0xA50AB56B,
01036       0x35B5A8FA, 0x42B2986C, 0xDBBBC9D6, 0xACBCF940, 0x32D86CE3, 0x45DF5C75, 0xDCD60DCF, 0xABD13D59,
01037       0x26D930AC, 0x51DE003A, 0xC8D75180, 0xBFD06116, 0x21B4F4B5, 0x56B3C423, 0xCFBA9599, 0xB8BDA50F,
01038       0x2802B89E, 0x5F058808, 0xC60CD9B2, 0xB10BE924, 0x2F6F7C87, 0x58684C11, 0xC1611DAB, 0xB6662D3D,
01039       0x76DC4190, 0x01DB7106, 0x98D220BC, 0xEFD5102A, 0x71B18589, 0x06B6B51F, 0x9FBFE4A5, 0xE8B8D433,
01040       0x7807C9A2, 0x0F00F934, 0x9609A88E, 0xE10E9818, 0x7F6A0DBB, 0x086D3D2D, 0x91646C97, 0xE6635C01,
01041       0x6B6B51F4, 0x1C6C6162, 0x856530D8, 0xF262004E, 0x6C0695ED, 0x1B01A57B, 0x8208F4C1, 0xF50FC457,
01042       0x65B0D9C6, 0x12B7E950, 0x8BBEB8EA, 0xFCB9887C, 0x62DD1DDF, 0x15DA2D49, 0x8CD37CF3, 0xFBD44C65,
01043       0x4DB26158, 0x3AB551CE, 0xA3BC0074, 0xD4BB30E2, 0x4ADFA541, 0x3DD895D7, 0xA4D1C46D, 0xD3D6F4FB,
01044       0x4369E96A, 0x346ED9FC, 0xAD678846, 0xDA60B8D0, 0x44042D73, 0x33031DE5, 0xAA0A4C5F, 0xDD0D7CC9,
01045       0x5005713C, 0x270241AA, 0xBE0B1010, 0xC90C2086, 0x5768B525, 0x206F85B3, 0xB966D409, 0xCE61E49F,
01046       0x5EDEF90E, 0x29D9C998, 0xB0D09822, 0xC7D7A8B4, 0x59B33D17, 0x2EB40D81, 0xB7BD5C3B, 0xC0BA6CAD,
01047       0xEDB88320, 0x9ABFB3B6, 0x03B6E20C, 0x74B1D29A, 0xEAD54739, 0x9DD277AF, 0x04DB2615, 0x73DC1683,
01048       0xE3630B12, 0x94643B84, 0x0D6D6A3E, 0x7A6A5AA8, 0xE40ECF0B, 0x9309FF9D, 0x0A00AE27, 0x7D079EB1,
01049       0xF00F9344, 0x8708A3D2, 0x1E01F268, 0x6906C2FE, 0xF762575D, 0x806567CB, 0x196C3671, 0x6E6B06E7,
01050       0xFED41B76, 0x89D32BE0, 0x10DA7A5A, 0x67DD4ACC, 0xF9B9DF6F, 0x8EBEEFF9, 0x17B7BE43, 0x60B08ED5,
01051       0xD6D6A3E8, 0xA1D1937E, 0x38D8C2C4, 0x4FDFF252, 0xD1BB67F1, 0xA6BC5767, 0x3FB506DD, 0x48B2364B,
01052       0xD80D2BDA, 0xAF0A1B4C, 0x36034AF6, 0x41047A60, 0xDF60EFC3, 0xA867DF55, 0x316E8EEF, 0x4669BE79,
01053       0xCB61B38C, 0xBC66831A, 0x256FD2A0, 0x5268E236, 0xCC0C7795, 0xBB0B4703, 0x220216B9, 0x5505262F,
01054       0xC5BA3BBE, 0xB2BD0B28, 0x2BB45A92, 0x5CB36A04, 0xC2D7FFA7, 0xB5D0CF31, 0x2CD99E8B, 0x5BDEAE1D,
01055       0x9B64C2B0, 0xEC63F226, 0x756AA39C, 0x026D930A, 0x9C0906A9, 0xEB0E363F, 0x72076785, 0x05005713,
01056       0x95BF4A82, 0xE2B87A14, 0x7BB12BAE, 0x0CB61B38, 0x92D28E9B, 0xE5D5BE0D, 0x7CDCEFB7, 0x0BDBDF21,
01057       0x86D3D2D4, 0xF1D4E242, 0x68DDB3F8, 0x1FDA836E, 0x81BE16CD, 0xF6B9265B, 0x6FB077E1, 0x18B74777,
01058       0x88085AE6, 0xFF0F6A70, 0x66063BCA, 0x11010B5C, 0x8F659EFF, 0xF862AE69, 0x616BFFD3, 0x166CCF45,
01059       0xA00AE278, 0xD70DD2EE, 0x4E048354, 0x3903B3C2, 0xA7672661, 0xD06016F7, 0x4969474D, 0x3E6E77DB,
01060       0xAED16A4A, 0xD9D65ADC, 0x40DF0B66, 0x37D83BF0, 0xA9BCAE53, 0xDEBB9EC5, 0x47B2CF7F, 0x30B5FFE9,
01061       0xBDBDF21C, 0xCABAC28A, 0x53B39330, 0x24B4A3A6, 0xBAD03605, 0xCDD70693, 0x54DE5729, 0x23D967BF,
01062       0xB3667A2E, 0xC4614AB8, 0x5D681B02, 0x2A6F2B94, 0xB40BBE37, 0xC30C8EA1, 0x5A05DF1B, 0x2D02EF8D
01063    };
01064
01065    unsigned int crc = ~0u;
01066    int i;
01067    for (i=0; i < len; ++i)
01068       crc = (crc » 8) ^ crc_table[buffer[i] ^ (crc & 0xff)];
01069    return ~crc;
01070 #endif
01071 }
01072
01073 #define stbiw__wpng4(o,a,b,c,d)
      ((o)[0]=STBIW_UCHAR(a),(o)[1]=STBIW_UCHAR(b),(o)[2]=STBIW_UCHAR(c),(o)[3]=STBIW_UCHAR(d),(o)+=4)
01074 #define stbiw__wp32(data,v) stbiw__wpng4(data, (v)»24,(v)»16,(v)»8,(v));
01075 #define stbiw__wptag(data,s) stbiw__wpng4(data, s[0],s[1],s[2],s[3])
01076
01077 static void stbiw__wpcrc(unsigned char **data, int len)
01078 {
01079    unsigned int crc = stbiw__crc32(*data - len - 4, len+4);
01080    stbiw__wp32(*data, crc);
01081 }
01082
01083 static unsigned char stbiw__paeth(int a, int b, int c)
01084 {
```

```
01085    int p = a + b - c, pa = abs(p-a), pb = abs(p-b), pc = abs(p-c);
01086    if (pa <= pb && pa <= pc) return STBIW_UCHAR(a);
01087    if (pb <= pc) return STBIW_UCHAR(b);
01088    return STBIW_UCHAR(c);
01089 }
01090
01091 // @OPTIMIZE: provide an option that always forces left-predict or paeth predict
01092 static void stbiw__encode_png_line(unsigned char *pixels, int stride_bytes, int width, int height, int
      y, int n, int filter_type, signed char *line_buffer)
01093 {
01094    static int mapping[] = { 0,1,2,3,4 };
01095    static int firstmap[] = { 0,1,0,5,6 };
01096    int *mymap = (y != 0) ? mapping : firstmap;
01097    int i;
01098    int type = mymap[filter_type];
01099    unsigned char *z = pixels + stride_bytes * (stbi__flip_vertically_on_write ? height-1-y : y);
01100    int signed_stride = stbi__flip_vertically_on_write ? -stride_bytes : stride_bytes;
01101
01102    if (type==0) {
01103       memcpy(line_buffer, z, width*n);
01104       return;
01105    }
01106
01107    // first loop isn't optimized since it's just one pixel
01108    for (i = 0; i < n; ++i) {
01109       switch (type) {
01110          case 1: line_buffer[i] = z[i]; break;
01111          case 2: line_buffer[i] = z[i] - z[i-signed_stride]; break;
01112          case 3: line_buffer[i] = z[i] - (z[i-signed_stride]>>1); break;
01113          case 4: line_buffer[i] = (signed char) (z[i] - stbiw__paeth(0,z[i-signed_stride],0)); break;
01114          case 5: line_buffer[i] = z[i]; break;
01115          case 6: line_buffer[i] = z[i]; break;
01116       }
01117    }
01118    switch (type) {
01119       case 1: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - z[i-n]; break;
01120       case 2: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - z[i-signed_stride]; break;
01121       case 3: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - ((z[i-n] + z[i-signed_stride])>>1);
      break;
01122       case 4: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - stbiw__paeth(z[i-n],
      z[i-signed_stride], z[i-signed_stride-n]); break;
01123       case 5: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - (z[i-n]>>1); break;
01124       case 6: for (i=n; i < width*n; ++i) line_buffer[i] = z[i] - stbiw__paeth(z[i-n], 0,0); break;
01125    }
01126 }
01127
01128 STBIWDEF unsigned char *stbi_write_png_to_mem(const unsigned char *pixels, int stride_bytes, int x,
      int y, int n, int *out_len)
01129 {
01130    int force_filter = stbi_write_force_png_filter;
01131    int ctype[5] = { -1, 0, 4, 2, 6 };
01132    unsigned char sig[8] = { 137,80,78,71,13,10,26,10 };
01133    unsigned char *out,*o, *filt, *zlib;
01134    signed char *line_buffer;
01135    int j,zlen;
01136
01137    if (stride_bytes == 0)
01138       stride_bytes = x * n;
01139
01140    if (force_filter >= 5) {
01141       force_filter = -1;
01142    }
01143
01144    filt = (unsigned char *) STBIW_MALLOC((x*n+1) * y); if (!filt) return 0;
01145    line_buffer = (signed char *) STBIW_MALLOC(x * n); if (!line_buffer) { STBIW_FREE(filt); return 0;
      }
01146    for (j=0; j < y; ++j) {
01147       int filter_type;
01148       if (force_filter > -1) {
01149          filter_type = force_filter;
01150          stbiw__encode_png_line((unsigned char*)(pixels), stride_bytes, x, y, j, n, force_filter,
      line_buffer);
01151       } else { // Estimate the best filter by running through all of them:
01152          int best_filter = 0, best_filter_val = 0x7fffffff, est, i;
01153          for (filter_type = 0; filter_type < 5; filter_type++) {
01154             stbiw__encode_png_line((unsigned char*)(pixels), stride_bytes, x, y, j, n, filter_type,
      line_buffer);
01155
01156             // Estimate the entropy of the line using this filter; the less, the better.
01157             est = 0;
01158             for (i = 0; i < x*n; ++i) {
01159                est += abs((signed char) line_buffer[i]);
01160             }
01161             if (est < best_filter_val) {
01162                best_filter_val = est;
01163                best_filter = filter_type;
01164             }
```

```
01165          }
01166          if (filter_type != best_filter) {  // If the last iteration already got us the best filter,
    don't redo it
01167             stbiw__encode_png_line((unsigned char*)(pixels), stride_bytes, x, y, j, n, best_filter,
    line_buffer);
01168             filter_type = best_filter;
01169          }
01170       }
01171       // when we get here, filter_type contains the filter type, and line_buffer contains the data
01172       filt[j*(x*n+1)] = (unsigned char) filter_type;
01173       STBIW_MEMMOVE(filt+j*(x*n+1)+1, line_buffer, x*n);
01174    }
01175    STBIW_FREE(line_buffer);
01176    zlib = stbi_zlib_compress(filt, y*( x*n+1), &zlen, stbi_write_png_compression_level);
01177    STBIW_FREE(filt);
01178    if (!zlib) return 0;
01179
01180    // each tag requires 12 bytes of overhead
01181    out = (unsigned char *) STBIW_MALLOC(8 + 12+13 + 12+zlen + 12);
01182    if (!out) return 0;
01183    *out_len = 8 + 12+13 + 12+zlen + 12;
01184
01185    o=out;
01186    STBIW_MEMMOVE(o,sig,8); o+= 8;
01187    stbiw__wp32(o, 13); // header length
01188    stbiw__wptag(o, "IHDR");
01189    stbiw__wp32(o, x);
01190    stbiw__wp32(o, y);
01191    *o++ = 8;
01192    *o++ = STBIW_UCHAR(ctype[n]);
01193    *o++ = 0;
01194    *o++ = 0;
01195    *o++ = 0;
01196    stbiw__wpcrc(&o,13);
01197
01198    stbiw__wp32(o, zlen);
01199    stbiw__wptag(o, "IDAT");
01200    STBIW_MEMMOVE(o, zlib, zlen);
01201    o += zlen;
01202    STBIW_FREE(zlib);
01203    stbiw__wpcrc(&o, zlen);
01204
01205    stbiw__wp32(o,0);
01206    stbiw__wptag(o, "IEND");
01207    stbiw__wpcrc(&o,0);
01208
01209    STBIW_ASSERT(o == out + *out_len);
01210
01211    return out;
01212 }
01213
01214 #ifndef STBI_WRITE_NO_STDIO
01215 STBIWDEF int stbi_write_png(char const *filename, int x, int y, int comp, const void *data, int
    stride_bytes)
01216 {
01217    FILE *f;
01218    int len;
01219    unsigned char *png = stbi_write_png_to_mem((const unsigned char *) data, stride_bytes, x, y, comp,
    &len);
01220    if (png == NULL) return 0;
01221
01222    f = stbiw__fopen(filename, "wb");
01223    if (!f) { STBIW_FREE(png); return 0; }
01224    fwrite(png, 1, len, f);
01225    fclose(f);
01226    STBIW_FREE(png);
01227    return 1;
01228 }
01229 #endif
01230
01231 STBIWDEF int stbi_write_png_to_func(stbi_write_func *func, void *context, int x, int y, int comp,
    const void *data, int stride_bytes)
01232 {
01233    int len;
01234    unsigned char *png = stbi_write_png_to_mem((const unsigned char *) data, stride_bytes, x, y, comp,
    &len);
01235    if (png == NULL) return 0;
01236    func(context, png, len);
01237    STBIW_FREE(png);
01238    return 1;
01239 }
01240
01241
01242 /* ***************************************************************************
01243  *
01244  * JPEG writer
01245  *
```

```
01246  * This is based on Jon Olick's jo_jpeg.cpp:
01247  * public domain Simple, Minimalistic JPEG writer - http://www.jonolick.com/code.html
01248  */
01249
01250 static const unsigned char stbiw__jpg_ZigZag[] = {
      0,1,5,6,14,15,27,28,2,4,7,13,16,26,29,42,3,8,12,17,25,30,41,43,9,11,18,
01251
      24,31,40,44,53,10,19,23,32,39,45,52,54,20,22,33,38,46,51,55,60,21,34,37,47,50,56,59,61,35,36,48,49,57,58,62,63
      };
01252
01253 static void stbiw__jpg_writeBits(stbi__write_context *s, int *bitBufP, int *bitCntP, const unsigned
      short *bs) {
01254    int bitBuf = *bitBufP, bitCnt = *bitCntP;
01255    bitCnt += bs[1];
01256    bitBuf |= bs[0] << (24 - bitCnt);
01257    while(bitCnt >= 8) {
01258       unsigned char c = (bitBuf >> 16) & 255;
01259       stbiw__putc(s, c);
01260       if(c == 255) {
01261          stbiw__putc(s, 0);
01262       }
01263       bitBuf <<= 8;
01264       bitCnt -= 8;
01265    }
01266    *bitBufP = bitBuf;
01267    *bitCntP = bitCnt;
01268 }
01269
01270 static void stbiw__jpg_DCT(float *d0p, float *d1p, float *d2p, float *d3p, float *d4p, float *d5p,
      float *d6p, float *d7p) {
01271    float d0 = *d0p, d1 = *d1p, d2 = *d2p, d3 = *d3p, d4 = *d4p, d5 = *d5p, d6 = *d6p, d7 = *d7p;
01272    float z1, z2, z3, z4, z5, z11, z13;
01273
01274    float tmp0 = d0 + d7;
01275    float tmp7 = d0 - d7;
01276    float tmp1 = d1 + d6;
01277    float tmp6 = d1 - d6;
01278    float tmp2 = d2 + d5;
01279    float tmp5 = d2 - d5;
01280    float tmp3 = d3 + d4;
01281    float tmp4 = d3 - d4;
01282
01283    // Even part
01284    float tmp10 = tmp0 + tmp3;   // phase 2
01285    float tmp13 = tmp0 - tmp3;
01286    float tmp11 = tmp1 + tmp2;
01287    float tmp12 = tmp1 - tmp2;
01288
01289    d0 = tmp10 + tmp11;       // phase 3
01290    d4 = tmp10 - tmp11;
01291
01292    z1 = (tmp12 + tmp13) * 0.707106781f; // c4
01293    d2 = tmp13 + z1;       // phase 5
01294    d6 = tmp13 - z1;
01295
01296    // Odd part
01297    tmp10 = tmp4 + tmp5;       // phase 2
01298    tmp11 = tmp5 + tmp6;
01299    tmp12 = tmp6 + tmp7;
01300
01301    // The rotator is modified from fig 4-8 to avoid extra negations.
01302    z5 = (tmp10 - tmp12) * 0.382683433f; // c6
01303    z2 = tmp10 * 0.541196100f + z5; // c2-c6
01304    z4 = tmp12 * 1.306562965f + z5; // c2+c6
01305    z3 = tmp11 * 0.707106781f; // c4
01306
01307    z11 = tmp7 + z3;      // phase 5
01308    z13 = tmp7 - z3;
01309
01310    *d5p = z13 + z2;         // phase 6
01311    *d3p = z13 - z2;
01312    *d1p = z11 + z4;
01313    *d7p = z11 - z4;
01314
01315    *d0p = d0;  *d2p = d2;  *d4p = d4;  *d6p = d6;
01316 }
01317
01318 static void stbiw__jpg_calcBits(int val, unsigned short bits[2]) {
01319    int tmp1 = val < 0 ? -val : val;
01320    val = val < 0 ? val-1 : val;
01321    bits[1] = 1;
01322    while(tmp1 >>= 1) {
01323       ++bits[1];
01324    }
01325    bits[0] = val & ((1<<bits[1])-1);
01326 }
01327
```

```
01328 static int stbiw__jpg_processDU(stbi__write_context *s, int *bitBuf, int *bitCnt, float *CDU, int
      du_stride, float *fdtbl, int DC, const unsigned short HTDC[256][2], const unsigned short HTAC[256][2])
      {
01329    const unsigned short EOB[2] = { HTAC[0x00][0], HTAC[0x00][1] };
01330    const unsigned short M16zeroes[2] = { HTAC[0xF0][0], HTAC[0xF0][1] };
01331    int dataOff, i, j, n, diff, end0pos, x, y;
01332    int DU[64];
01333
01334    // DCT rows
01335    for(dataOff=0, n=du_stride*8; dataOff<n; dataOff+=du_stride) {
01336       stbiw__jpg_DCT(&CDU[dataOff], &CDU[dataOff+1], &CDU[dataOff+2], &CDU[dataOff+3],
      &CDU[dataOff+4], &CDU[dataOff+5], &CDU[dataOff+6], &CDU[dataOff+7]);
01337    }
01338    // DCT columns
01339    for(dataOff=0; dataOff<8; ++dataOff) {
01340       stbiw__jpg_DCT(&CDU[dataOff], &CDU[dataOff+du_stride], &CDU[dataOff+du_stride*2],
      &CDU[dataOff+du_stride*3], &CDU[dataOff+du_stride*4],
01341                      &CDU[dataOff+du_stride*5], &CDU[dataOff+du_stride*6], &CDU[dataOff+du_stride*7]);
01342    }
01343    // Quantize/descale/zigzag the coefficients
01344    for(y = 0, j=0; y < 8; ++y) {
01345       for(x = 0; x < 8; ++x,++j) {
01346          float v;
01347          i = y*du_stride+x;
01348          v = CDU[i]*fdtbl[j];
01349          // DU[stbiw__jpg_ZigZag[j]] = (int)(v < 0 ? ceilf(v - 0.5f) : floorf(v + 0.5f));
01350          // ceilf() and floorf() are C99, not C89, but I /think/ they're not needed here anyway?
01351          DU[stbiw__jpg_ZigZag[j]] = (int)(v < 0 ? v - 0.5f : v + 0.5f);
01352       }
01353    }
01354
01355    // Encode DC
01356    diff = DU[0] - DC;
01357    if (diff == 0) {
01358       stbiw__jpg_writeBits(s, bitBuf, bitCnt, HTDC[0]);
01359    } else {
01360       unsigned short bits[2];
01361       stbiw__jpg_calcBits(diff, bits);
01362       stbiw__jpg_writeBits(s, bitBuf, bitCnt, HTDC[bits[1]]);
01363       stbiw__jpg_writeBits(s, bitBuf, bitCnt, bits);
01364    }
01365    // Encode ACs
01366    end0pos = 63;
01367    for(; (end0pos>0)&&(DU[end0pos]==0); --end0pos) {
01368    }
01369    // end0pos = first element in reverse order !=0
01370    if(end0pos == 0) {
01371       stbiw__jpg_writeBits(s, bitBuf, bitCnt, EOB);
01372       return DU[0];
01373    }
01374    for(i = 1; i <= end0pos; ++i) {
01375       int startpos = i;
01376       int nrzeroes;
01377       unsigned short bits[2];
01378       for (; DU[i]==0 && i<=end0pos; ++i) {
01379       }
01380       nrzeroes = i-startpos;
01381       if ( nrzeroes >= 16 ) {
01382          int lng = nrzeroes»4;
01383          int nrmarker;
01384          for (nrmarker=1; nrmarker <= lng; ++nrmarker)
01385             stbiw__jpg_writeBits(s, bitBuf, bitCnt, M16zeroes);
01386          nrzeroes &= 15;
01387       }
01388       stbiw__jpg_calcBits(DU[i], bits);
01389       stbiw__jpg_writeBits(s, bitBuf, bitCnt, HTAC[(nrzeroes«4)+bits[1]]);
01390       stbiw__jpg_writeBits(s, bitBuf, bitCnt, bits);
01391    }
01392    if(end0pos != 63) {
01393       stbiw__jpg_writeBits(s, bitBuf, bitCnt, EOB);
01394    }
01395    return DU[0];
01396 }
01397
01398 static int stbi_write_jpg_core(stbi__write_context *s, int width, int height, int comp, const void*
      data, int quality) {
01399    // Constants that don't pollute global namespace
01400    static const unsigned char std_dc_luminance_nrcodes[] = {0,0,1,5,1,1,1,1,1,1,0,0,0,0,0,0,0};
01401    static const unsigned char std_dc_luminance_values[] = {0,1,2,3,4,5,6,7,8,9,10,11};
01402    static const unsigned char std_ac_luminance_nrcodes[] = {0,0,2,1,3,3,2,4,3,5,5,4,4,0,0,1,0x7d};
01403    static const unsigned char std_ac_luminance_values[] = {
01404
      0x01,0x02,0x03,0x00,0x04,0x11,0x05,0x12,0x21,0x31,0x41,0x06,0x13,0x51,0x61,0x07,0x22,0x71,0x14,0x32,0x81,0x91,0xa1,0x08,
01405
      0x23,0x42,0xb1,0xc1,0x15,0x52,0xd1,0xf0,0x24,0x33,0x62,0x72,0x82,0x09,0x0a,0x16,0x17,0x18,0x19,0x1a,0x25,0x26,0x27,0x28,
01406
      0x29,0x2a,0x34,0x35,0x36,0x37,0x38,0x39,0x3a,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x53,0x54,0x55,0x56,0x57,0x58,0x59,
```

```
01407
      0x5a,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6a,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7a,0x83,0x84,0x85,0x86,0x87,0x88,0x89,
01408
      0x8a,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xb2,0xb3,0xb4,0xb5,0xb6,
01409
      0xb7,0xb8,0xb9,0xba,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xca,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,0xe1,0xe2,
01410       0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa
01411    };
01412    static const unsigned char std_dc_chrominance_nrcodes[] = {0,0,3,1,1,1,1,1,1,1,1,1,0,0,0,0,0};
01413    static const unsigned char std_dc_chrominance_values[] = {0,1,2,3,4,5,6,7,8,9,10,11};
01414    static const unsigned char std_ac_chrominance_nrcodes[] = {0,0,2,1,2,4,4,3,4,7,5,4,4,0,1,2,0x77};
01415    static const unsigned char std_ac_chrominance_values[] = {
01416
      0x00,0x01,0x02,0x03,0x11,0x04,0x05,0x21,0x31,0x06,0x12,0x41,0x51,0x07,0x61,0x71,0x13,0x22,0x32,0x81,0x08,0x14,0x42,0x91,
01417
      0xa1,0xb1,0xc1,0x09,0x23,0x33,0x52,0xf0,0x15,0x62,0x72,0xd1,0x0a,0x16,0x24,0x34,0xe1,0x25,0xf1,0x17,0x18,0x19,0x1a,0x26,
01418
      0x27,0x28,0x29,0x2a,0x35,0x36,0x37,0x38,0x39,0x3a,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x53,0x54,0x55,0x56,0x57,0x58,
01419
      0x59,0x5a,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6a,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7a,0x82,0x83,0x84,0x85,0x86,0x87,
01420
      0x88,0x89,0x8a,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xb2,0xb3,0xb4,
01421
      0xb5,0xb6,0xb7,0xb8,0xb9,0xba,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xca,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,
01422       0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa
01423    };
01424    // Huffman tables
01425    static const unsigned short YDC_HT[256][2] = {
      {0,2},{2,3},{3,3},{4,3},{5,3},{6,3},{14,4},{30,5},{62,6},{126,7},{254,8},{510,9}};
01426    static const unsigned short UVDC_HT[256][2] = {
      {0,2},{1,2},{2,2},{6,3},{14,4},{30,5},{62,6},{126,7},{254,8},{510,9},{1022,10},{2046,11}};
01427    static const unsigned short YAC_HT[256][2] = {
01428
      {10,4},{0,2},{1,2},{4,3},{11,4},{26,5},{120,7},{248,8},{1014,10},{65410,16},{65411,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},{0,
01429
      {12,4},{27,5},{121,7},{502,9},{2038,11},{65412,16},{65413,16},{65414,16},{65415,16},{65416,16},{0,0},{0,0},{0,0},{0,0},{
01430
      {28,5},{249,8},{1015,10},{4084,12},{65417,16},{65418,16},{65419,16},{65420,16},{65421,16},{65422,16},{0,0},{0,0},{0,0},{
01431
      {58,6},{503,9},{4085,12},{65423,16},{65424,16},{65425,16},{65426,16},{65427,16},{65428,16},{65429,16},{0,0},{0,0},{0,0},
01432
      {59,6},{1016,10},{65430,16},{65431,16},{65432,16},{65433,16},{65434,16},{65435,16},{65436,16},{65437,16},{0,0},{0,0},{0,
01433
      {122,7},{2039,11},{65438,16},{65439,16},{65440,16},{65441,16},{65442,16},{65443,16},{65444,16},{65445,16},{0,0},{0,0},{
01434
      {123,7},{4086,12},{65446,16},{65447,16},{65448,16},{65449,16},{65450,16},{65451,16},{65452,16},{65453,16},{0,0},{0,0},{
01435
      {250,8},{4087,12},{65454,16},{65455,16},{65456,16},{65457,16},{65458,16},{65459,16},{65460,16},{65461,16},{0,0},{0,0},{
01436
      {504,9},{32704,15},{65462,16},{65463,16},{65464,16},{65465,16},{65466,16},{65467,16},{65468,16},{65469,16},{0,0},{0,0},
01437
      {505,9},{65470,16},{65471,16},{65472,16},{65473,16},{65474,16},{65475,16},{65476,16},{65477,16},{65478,16},{0,0},{0,0},
01438
      {506,9},{65479,16},{65480,16},{65481,16},{65482,16},{65483,16},{65484,16},{65485,16},{65486,16},{65487,16},{0,0},{0,0},
01439
      {1017,10},{65488,16},{65489,16},{65490,16},{65491,16},{65492,16},{65493,16},{65494,16},{65495,16},{65496,16},{0,0},{0,0
01440
      {1018,10},{65497,16},{65498,16},{65499,16},{65500,16},{65501,16},{65502,16},{65503,16},{65504,16},{65505,16},{0,0},{0,0
01441
      {2040,11},{65506,16},{65507,16},{65508,16},{65509,16},{65510,16},{65511,16},{65512,16},{65513,16},{65514,16},{0,0},{0,0
01442
      {65515,16},{65516,16},{65517,16},{65518,16},{65519,16},{65520,16},{65521,16},{65522,16},{65523,16},{65524,16},{0,0},{0,0
01443
      {2041,11},{65525,16},{65526,16},{65527,16},{65528,16},{65529,16},{65530,16},{65531,16},{65532,16},{65533,16},{65534,16},
01444    };
01445    static const unsigned short UVAC_HT[256][2] = {
01446
      {0,2},{1,2},{4,3},{10,4},{24,5},{25,5},{56,6},{120,7},{500,9},{1014,10},{4084,12},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},
01447
      {11,4},{57,6},{246,8},{501,9},{2038,11},{4085,12},{65416,16},{65417,16},{65418,16},{65419,16},{0,0},{0,0},{0,0},{0,0},{
01448
      {26,5},{247,8},{1015,10},{4086,12},{32706,15},{65420,16},{65421,16},{65422,16},{65423,16},{65424,16},{0,0},{0,0},{0,0},
01449
      {27,5},{248,8},{1016,10},{4087,12},{65425,16},{65426,16},{65427,16},{65428,16},{65429,16},{65430,16},{0,0},{0,0},{0,0},
01450
      {58,6},{502,9},{65431,16},{65432,16},{65433,16},{65434,16},{65435,16},{65436,16},{65437,16},{65438,16},{0,0},{0,0},{0,0
01451
      {59,6},{1017,10},{65439,16},{65440,16},{65441,16},{65442,16},{65443,16},{65444,16},{65445,16},{65446,16},{0,0},{0,0},{0,
01452
      {121,7},{2039,11},{65447,16},{65448,16},{65449,16},{65450,16},{65451,16},{65452,16},{65453,16},{65454,16},{0,0},{0,0},{
01453
      {122,7},{2040,11},{65455,16},{65456,16},{65457,16},{65458,16},{65459,16},{65460,16},{65461,16},{65462,16},{0,0},{0,0},{
01454
      {249,8},{65463,16},{65464,16},{65465,16},{65466,16},{65467,16},{65468,16},{65469,16},{65470,16},{65471,16},{0,0},{0,0},
01455
      {503,9},{65472,16},{65473,16},{65474,16},{65475,16},{65476,16},{65477,16},{65478,16},{65479,16},{65480,16},{0,0},{0,0},
01456
```

```
      {504,9},{65481,16},{65482,16},{65483,16},{65484,16},{65485,16},{65486,16},{65487,16},{65488,16},{65489,16},{0,0},{0,0},
01457
      {505,9},{65490,16},{65491,16},{65492,16},{65493,16},{65494,16},{65495,16},{65496,16},{65497,16},{65498,16},{0,0},{0,0},
01458
      {506,9},{65499,16},{65500,16},{65501,16},{65502,16},{65503,16},{65504,16},{65505,16},{65506,16},{65507,16},{0,0},{0,0},
01459
      {2041,11},{65508,16},{65509,16},{65510,16},{65511,16},{65512,16},{65513,16},{65514,16},{65515,16},{65516,16},{0,0},{0,0
01460
      {16352,14},{65517,16},{65518,16},{65519,16},{65520,16},{65521,16},{65522,16},{65523,16},{65524,16},{65525,16},{0,0},{0,
01461
      {1018,10},{32707,15},{65526,16},{65527,16},{65528,16},{65529,16},{65530,16},{65531,16},{65532,16},{65533,16},{65534,16},
01462    };
01463    static const int YQT[] =
      {16,11,10,16,24,40,51,61,12,12,14,19,26,58,60,55,14,13,16,24,40,57,69,56,14,17,22,29,51,87,80,62,18,22,
01464
      37,56,68,109,103,77,24,35,55,64,81,104,113,92,49,64,78,87,103,121,120,101,72,92,95,98,112,100,103,99};
01465    static const int UVQT[] =
      {17,18,24,47,99,99,99,99,18,21,26,66,99,99,99,99,24,26,56,99,99,99,99,99,47,66,99,99,99,99,99,99,
01466
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99};
01467    static const float aasf[] = { 1.0f * 2.828427125f, 1.387039845f * 2.828427125f, 1.306562965f *
      2.828427125f, 1.175875602f * 2.828427125f,
01468                                  1.0f * 2.828427125f, 0.785694958f * 2.828427125f, 0.541196100f *
      2.828427125f, 0.275899379f * 2.828427125f };
01469
01470    int row, col, i, k, subsample;
01471    float fdtbl_Y[64], fdtbl_UV[64];
01472    unsigned char YTable[64], UVTable[64];
01473
01474    if(!data || !width || !height || comp > 4 || comp < 1) {
01475       return 0;
01476    }
01477
01478    quality = quality ? quality : 90;
01479    subsample = quality <= 90 ? 1 : 0;
01480    quality = quality < 1 ? 1 : quality > 100 ? 100 : quality;
01481    quality = quality < 50 ? 5000 / quality : 200 - quality * 2;
01482
01483    for(i = 0; i < 64; ++i) {
01484       int uvti, yti = (YQT[i]*quality+50)/100;
01485       YTable[stbiw__jpg_ZigZag[i]] = (unsigned char) (yti < 1 ? 1 : yti > 255 ? 255 : yti);
01486       uvti = (UVQT[i]*quality+50)/100;
01487       UVTable[stbiw__jpg_ZigZag[i]] = (unsigned char) (uvti < 1 ? 1 : uvti > 255 ? 255 : uvti);
01488    }
01489
01490    for(row = 0, k = 0; row < 8; ++row) {
01491       for(col = 0; col < 8; ++col, ++k) {
01492          fdtbl_Y[k]  = 1 / (YTable [stbiw__jpg_ZigZag[k]] * aasf[row] * aasf[col]);
01493          fdtbl_UV[k] = 1 / (UVTable[stbiw__jpg_ZigZag[k]] * aasf[row] * aasf[col]);
01494       }
01495    }
01496
01497    // Write Headers
01498    {
01499       static const unsigned char head0[] = {
      0xFF,0xD8,0xFF,0xE0,0,0x10,'J','F','I','F',0,1,1,0,0,1,0,1,0,0,0xFF,0xDB,0,0x84,0 };
01500       static const unsigned char head2[] = { 0xFF,0xDA,0,0xC,3,1,0,2,0x11,3,0x11,0,0x3F,0 };
01501       const unsigned char head1[] = { 0xFF,0xC0,0,0x11,8,(unsigned
      char)(height>>8),STBIW_UCHAR(height),(unsigned char)(width>>8),STBIW_UCHAR(width),
01502                                       3,1,(unsigned
      char)(subsample?0x22:0x11),0,2,0x11,1,3,0x11,1,0xFF,0xC4,0x01,0xA2,0 };
01503       s->func(s->context, (void*)head0, sizeof(head0));
01504       s->func(s->context, (void*)YTable, sizeof(YTable));
01505       stbiw__putc(s, 1);
01506       s->func(s->context, UVTable, sizeof(UVTable));
01507       s->func(s->context, (void*)head1, sizeof(head1));
01508       s->func(s->context, (void*)(std_dc_luminance_nrcodes+1), sizeof(std_dc_luminance_nrcodes)-1);
01509       s->func(s->context, (void*)std_dc_luminance_values, sizeof(std_dc_luminance_values));
01510       stbiw__putc(s, 0x10); // HTYACinfo
01511       s->func(s->context, (void*)(std_ac_luminance_nrcodes+1), sizeof(std_ac_luminance_nrcodes)-1);
01512       s->func(s->context, (void*)std_ac_luminance_values, sizeof(std_ac_luminance_values));
01513       stbiw__putc(s, 1); // HTUDCinfo
01514       s->func(s->context, (void*)(std_dc_chrominance_nrcodes+1),
      sizeof(std_dc_chrominance_nrcodes)-1);
01515       s->func(s->context, (void*)std_dc_chrominance_values, sizeof(std_dc_chrominance_values));
01516       stbiw__putc(s, 0x11); // HTUACinfo
01517       s->func(s->context, (void*)(std_ac_chrominance_nrcodes+1),
      sizeof(std_ac_chrominance_nrcodes)-1);
01518       s->func(s->context, (void*)std_ac_chrominance_values, sizeof(std_ac_chrominance_values));
01519       s->func(s->context, (void*)head2, sizeof(head2));
01520    }
01521
01522    // Encode 8x8 macroblocks
01523    {
01524       static const unsigned short fillBits[] = {0x7F, 7};
01525       int DCY=0, DCU=0, DCV=0;
01526       int bitBuf=0, bitCnt=0;
```

```
01527       // comp == 2 is grey+alpha (alpha is ignored)
01528       int ofsG = comp > 2 ? 1 : 0, ofsB = comp > 2 ? 2 : 0;
01529       const unsigned char *dataR = (const unsigned char *)data;
01530       const unsigned char *dataG = dataR + ofsG;
01531       const unsigned char *dataB = dataR + ofsB;
01532       int x, y, pos;
01533       if(subsample) {
01534          for(y = 0; y < height; y += 16) {
01535             for(x = 0; x < width; x += 16) {
01536                float Y[256], U[256], V[256];
01537                for(row = y, pos = 0; row < y+16; ++row) {
01538                   // row >= height => use last input row
01539                   int clamped_row = (row < height) ? row : height - 1;
01540                   int base_p = (stbi__flip_vertically_on_write ? (height-1-clamped_row) :
      clamped_row)*width*comp;
01541                   for(col = x; col < x+16; ++col, ++pos) {
01542                      // if col >= width => use pixel from last input column
01543                      int p = base_p + ((col < width) ? col : (width-1))*comp;
01544                      float r = dataR[p], g = dataG[p], b = dataB[p];
01545                      Y[pos]= +0.29900f*r + 0.58700f*g + 0.11400f*b - 128;
01546                      U[pos]= -0.16874f*r - 0.33126f*g + 0.50000f*b;
01547                      V[pos]= +0.50000f*r - 0.41869f*g - 0.08131f*b;
01548                   }
01549                }
01550                DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, Y+0,   16, fdtbl_Y, DCY, YDC_HT,
      YAC_HT);
01551                DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, Y+8,   16, fdtbl_Y, DCY, YDC_HT,
      YAC_HT);
01552                DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, Y+128, 16, fdtbl_Y, DCY, YDC_HT,
      YAC_HT);
01553                DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, Y+136, 16, fdtbl_Y, DCY, YDC_HT,
      YAC_HT);
01554
01555                // subsample U,V
01556                {
01557                   float subU[64], subV[64];
01558                   int yy, xx;
01559                   for(yy = 0, pos = 0; yy < 8; ++yy) {
01560                      for(xx = 0; xx < 8; ++xx, ++pos) {
01561                         int j = yy*32+xx*2;
01562                         subU[pos] = (U[j+0] + U[j+1] + U[j+16] + U[j+17]) * 0.25f;
01563                         subV[pos] = (V[j+0] + V[j+1] + V[j+16] + V[j+17]) * 0.25f;
01564                      }
01565                   }
01566                   DCU = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, subU, 8, fdtbl_UV, DCU, UVDC_HT,
      UVAC_HT);
01567                   DCV = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, subV, 8, fdtbl_UV, DCV, UVDC_HT,
      UVAC_HT);
01568                }
01569             }
01570          }
01571       } else {
01572          for(y = 0; y < height; y += 8) {
01573             for(x = 0; x < width; x += 8) {
01574                float Y[64], U[64], V[64];
01575                for(row = y, pos = 0; row < y+8; ++row) {
01576                   // row >= height => use last input row
01577                   int clamped_row = (row < height) ? row : height - 1;
01578                   int base_p = (stbi__flip_vertically_on_write ? (height-1-clamped_row) :
      clamped_row)*width*comp;
01579                   for(col = x; col < x+8; ++col, ++pos) {
01580                      // if col >= width => use pixel from last input column
01581                      int p = base_p + ((col < width) ? col : (width-1))*comp;
01582                      float r = dataR[p], g = dataG[p], b = dataB[p];
01583                      Y[pos]= +0.29900f*r + 0.58700f*g + 0.11400f*b - 128;
01584                      U[pos]= -0.16874f*r - 0.33126f*g + 0.50000f*b;
01585                      V[pos]= +0.50000f*r - 0.41869f*g - 0.08131f*b;
01586                   }
01587                }
01588
01589                DCY = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, Y, 8, fdtbl_Y,  DCY, YDC_HT, YAC_HT);
01590                DCU = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, U, 8, fdtbl_UV, DCU, UVDC_HT, UVAC_HT);
01591                DCV = stbiw__jpg_processDU(s, &bitBuf, &bitCnt, V, 8, fdtbl_UV, DCV, UVDC_HT, UVAC_HT);
01592             }
01593          }
01594       }
01595
01596       // Do the bit alignment of the EOI marker
01597       stbiw__jpg_writeBits(s, &bitBuf, &bitCnt, fillBits);
01598    }
01599
01600    // EOI
01601    stbiw__putc(s, 0xFF);
01602    stbiw__putc(s, 0xD9);
01603
01604    return 1;
01605 }
```

```
01606
01607 STBIWDEF int stbi_write_jpg_to_func(stbi_write_func *func, void *context, int x, int y, int comp,
       const void *data, int quality)
01608 {
01609    stbi__write_context s = { 0 };
01610    stbi__start_write_callbacks(&s, func, context);
01611    return stbi_write_jpg_core(&s, x, y, comp, (void *) data, quality);
01612 }
01613
01614
01615 #ifndef STBI_WRITE_NO_STDIO
01616 STBIWDEF int stbi_write_jpg(char const *filename, int x, int y, int comp, const void *data, int
       quality)
01617 {
01618    stbi__write_context s = { 0 };
01619    if (stbi__start_write_file(&s,filename)) {
01620       int r = stbi_write_jpg_core(&s, x, y, comp, data, quality);
01621       stbi__end_write_file(&s);
01622       return r;
01623    } else
01624       return 0;
01625 }
01626 #endif
01627
01628 #endif // STB_IMAGE_WRITE_IMPLEMENTATION
01629
01630 /* Revision history
01631       1.16  (2021-07-11)
01632              make Deflate code emit uncompressed blocks when it would otherwise expand
01633              support writing BMPs with alpha channel
01634       1.15  (2020-07-13) unknown
01635       1.14  (2020-02-02) updated JPEG writer to downsample chroma channels
01636       1.13
01637       1.12
01638       1.11  (2019-08-11)
01639
01640       1.10  (2019-02-07)
01641              support utf8 filenames in Windows; fix warnings and platform ifdefs
01642       1.09  (2018-02-11)
01643              fix typo in zlib quality API, improve STB_I_W_STATIC in C++
01644       1.08  (2018-01-29)
01645              add stbi__flip_vertically_on_write, external zlib, zlib quality, choose PNG filter
01646       1.07  (2017-07-24)
01647              doc fix
01648       1.06 (2017-07-23)
01649              writing JPEG (using Jon Olick's code)
01650       1.05   ???
01651       1.04 (2017-03-03)
01652              monochrome BMP expansion
01653       1.03   ???
01654       1.02 (2016-04-02)
01655              avoid allocating large structures on the stack
01656       1.01 (2016-01-16)
01657              STBIW_REALLOC_SIZED: support allocators with no realloc support
01658              avoid race-condition in crc initialization
01659              minor compile issues
01660       1.00 (2015-09-14)
01661              installable file IO function
01662       0.99 (2015-09-13)
01663              warning fixes; TGA rle support
01664       0.98 (2015-04-08)
01665              added STBIW_MALLOC, STBIW_ASSERT etc
01666       0.97 (2015-01-18)
01667              fixed HDR asserts, rewrote HDR rle logic
01668       0.96 (2015-01-17)
01669              add HDR output
01670              fix monochrome BMP
01671       0.95 (2014-08-17)
01672              add monochrome TGA output
01673       0.94 (2014-05-31)
01674              rename private functions to avoid conflicts with stb_image.h
01675       0.93 (2014-05-27)
01676              warning fixes
01677       0.92 (2010-08-01)
01678              casts to unsigned char to fix warnings
01679       0.91 (2010-07-17)
01680              first public release
01681       0.90   first internal release
01682 */
01683
01684 /*
01685 ------------------------------------------------------------------------------
01686 This software is available under 2 licenses -- choose whichever you prefer.
01687 ------------------------------------------------------------------------------
01688 ALTERNATIVE A - MIT License
01689 Copyright (c) 2017 Sean Barrett
01690 Permission is hereby granted, free of charge, to any person obtaining a copy of
```

```
01691 this software and associated documentation files (the "Software"), to deal in
01692 the Software without restriction, including without limitation the rights to
01693 use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
01694 of the Software, and to permit persons to whom the Software is furnished to do
01695 so, subject to the following conditions:
01696 The above copyright notice and this permission notice shall be included in all
01697 copies or substantial portions of the Software.
01698 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
01699 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
01700 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
01701 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
01702 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
01703 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
01704 SOFTWARE.
01705 ------------------------------------------------------------------------------
01706 ALTERNATIVE B - Public Domain (www.unlicense.org)
01707 This is free and unencumbered software released into the public domain.
01708 Anyone is free to copy, modify, publish, use, compile, sell, or distribute this
01709 software, either in source code form or as a compiled binary, for any purpose,
01710 commercial or non-commercial, and by any means.
01711 In jurisdictions that recognize copyright laws, the author or authors of this
01712 software dedicate any and all copyright interest in the software to the public
01713 domain. We make this dedication for the benefit of the public at large and to
01714 the detriment of our heirs and successors. We intend this dedication to be an
01715 overt act of relinquishment in perpetuity of all present and future rights to
01716 this software under copyright law.
01717 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
01718 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
01719 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
01720 AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
01721 ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
01722 WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
01723 ------------------------------------------------------------------------------
01724 */
```

5.28 test.cpp File Reference

```
#include <iostream>
#include <string>
#include <array>
#include "stb/stb_image.h"
#include "stb/stb_image_write.h"
#include <filesystem>
```

Macros

- #define STB_IMAGE_IMPLEMENTATION
- #define STB_IMAGE_WRITE_IMPLEMENTATION
- #define OUTPUT_DIR "../Output/"

Functions

- int main ()

5.28.1 Macro Definition Documentation

5.28.1.1 OUTPUT_DIR

```
#define OUTPUT_DIR "../Output/"
```

5.28.1.2 STB_IMAGE_IMPLEMENTATION

```
#define STB_IMAGE_IMPLEMENTATION
```

5.28.1.3 STB_IMAGE_WRITE_IMPLEMENTATION

```
#define STB_IMAGE_WRITE_IMPLEMENTATION
```

5.28.2 Function Documentation

5.28.2.1 main()

```
int main ( )
```

5.29 unittest/Unittest.cpp File Reference

```
#include <cstring>
#include <cstdlib>
#include <cmath>
#include <vector>
#include <tuple>
#include "Unittest.h"
```

5.30 unittest/Unittest.h File Reference

```
#include "../filter/Filter.h"
#include "../filter/ImageFilter.h"
#include "../filter/VolumeFilter.h"
#include "../image/Image.h"
#include "../volume/Volume.h"
#include "../data/Data.h"
#include "../projection/Projection.h"
#include <iostream>
```

Classes

- class Unittest

5.31 Unittest.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009 #include "../filter/Filter.h"
00010 #include "../filter/ImageFilter.h"
00011 #include "../filter/VolumeFilter.h"
00012 #include "../image/Image.h"
00013 #include "../volume/Volume.h"
00014 #include "../data/Data.h"
00015 #include "../projection/Projection.h"
00016
00017 #include <iostream>
00018
00019 class Unittest
00020 {
00021     public:
00022     static std::tuple<int,int>  test_adjust_brightness();
00023     static std::tuple<int,int>  test_to_grayscale();
00024     static std::tuple<int,int>  test_color_balance();
00025     static std::tuple<int,int>  test_edge_detection();
00026     static std::tuple<int,int>  test_GaussianFilter();
00027     static std::tuple<int,int>  test_median_filter();
00028     static std::tuple<int,int>  test_histogram_equalization();
00029
00030     static std::tuple<int,int>  test_GaussianFilter3d(int kernelSize,double sigma);
00031     static std::tuple<int,int>  test_3D_median();
00032     static std::tuple<int,int>  test_3D_GaussianFilter();
00033
00034     static void test_MIP(bool specify = false, int min_z = 0, int max_z = 0);
00035     static void test_MinIP(bool specify = false, int min_z = 0, int max_z = 0);
00036     static void test_mean_AIP(bool specify = false, int min_z = 0, int max_z = 0);
00037     static void test_median_AIP(bool specify = false, int min_z = 0, int max_z = 0);
00038
00039     static void run_all();
00040
00041     private:
00042     static std::tuple<int,int> test_3D(int sign=0);
00043     static std::tuple<int,int>  test_2D(int sign);
00044
00045     int tests_passed = 0;
00046     int tests_failed = 0;
00047     int total_tests = 0;
00048 };
```

5.32 volume/Volume.cpp File Reference

```
#include "Volume.h"
#include <string>
```

5.33 volume/Volume.h File Reference

```
#include "../data/Data.h"
#include "../image/Image.h"
#include <vector>
```

Classes

- class Volume

  Class to represent 3D volumes.

5.34 Volume.h

Go to the documentation of this file.

```
00001
00007 #pragma once
00008
00009 #include "../data/Data.h"
00010 #include "../image/Image.h"
00011 #include "../data/Data.h"
00012 #include <vector>
00013
00017 class Volume : public Data
00018 {
00019 public:
00020
00021     // Constructors
00022     Volume(char* dir_name);
00023     Volume();
00024     Volume(unsigned char* data, int w, int h, int c, int num_files, char* name);
00025
00026     // destructor
00027     virtual ~Volume();
00028
00029     int get_x();
00030     int get_y();
00031     int get_z();
00032     int get_channels();
00033
00034     char* dir_name;
00035
00036
00037 protected:
00038
00039 private:
00040
00041
00042 };
```

Index

~Data
    Data, 10
~Filter
    Filter, 14
~Image
    Image, 17
~ImageFilter
    ImageFilter, 19
~Projection
    Projection, 22
~Volume
    Volume, 34
~VolumeFilter
    VolumeFilter, 35

adjust_brightness
    ImageFilter, 19

BoxFilter
    ImageFilter, 19
brightness
    Image.cpp, 41

c
    Data, 12
color_balance
    ImageFilter, 20

Data, 7
    ~Data, 10
    c, 12
    Data, 8–10
    data, 13
    file_name, 13
    file_read, 13
    get_size, 10
    h, 13
    input_file, 13
    is_dir, 13
    num_files, 13
    read_dir, 10
    read_file, 11
    set_input_file, 11
    size, 13
    update, 12
    w, 14
    write_file, 12
data
    Data, 13
Data.cpp
    OUTPUT_DIR, 37
    STB_IMAGE_IMPLEMENTATION, 37
    STB_IMAGE_WRITE_IMPLEMENTATION, 37
data/Data.cpp, 37
data/Data.h, 38
dir_name
    Volume, 34

edge_detection
    ImageFilter, 20
eof
    stbi_io_callbacks, 26

file_name
    Data, 13
file_read
    Data, 13
Filter, 14
    ~Filter, 14
filter
    Image.cpp, 41
filter/Filter.cpp, 39
filter/Filter.h, 39
filter/ImageFilter.cpp, 39
filter/ImageFilter.h, 39, 40
filter/VolumeFilter.cpp, 40
filter/VolumeFilter.h, 40, 41

GaussianFilter
    ImageFilter, 20
GaussianFilter3d
    VolumeFilter, 36
get_channels
    Image, 17
    Volume, 34
get_height
    Image, 17
get_size
    Data, 10
get_width
    Image, 17
get_x
    Volume, 34
get_y
    Volume, 34
get_z
    Volume, 34
grayscale
    Image.cpp, 41

h

Data, 13
histogram_equalization
ImageFilter, 20

Image, 15
~Image, 17
get_channels, 17
get_height, 17
get_width, 17
Image, 16, 17
Image.cpp
brightness, 41
filter, 41
grayscale, 41
image/Image.cpp, 41
image/Image.h, 42
ImageFilter, 18
~ImageFilter, 19
adjust_brightness, 19
BoxFilter, 19
color_balance, 20
edge_detection, 20
GaussianFilter, 20
histogram_equalization, 20
median_filter, 21
to_grayscale, 21
input_file
Data, 13
is_dir
Data, 13
is_int
main.cpp, 43

locating
Projection, 23

main
main.cpp, 43
minimal.cpp, 44
test.cpp, 178
main.cpp, 42
is_int, 43
main, 43
max_intensity_projection
Projection, 23
mean_avg_intensity_projection
Projection, 23
median3D_filter
VolumeFilter, 36
median_avg_intensity_projection
Projection, 24
median_filter
ImageFilter, 21
min_intensity_projection
Projection, 24
minimal.cpp, 43
main, 44
STB_IMAGE_IMPLEMENTATION, 43
STB_IMAGE_WRITE_IMPLEMENTATION, 43

num_files
Data, 13

OUTPUT_DIR
Data.cpp, 37
test.cpp, 177

Projection, 21
~Projection, 22
locating, 23
max_intensity_projection, 23
mean_avg_intensity_projection, 23
median_avg_intensity_projection, 24
min_intensity_projection, 24
projection/Projection.cpp, 44
projection/Projection.h, 44, 45

read
stbi_io_callbacks, 26
read_dir
Data, 10
read_file
Data, 11
run_all
Unittest, 28

set_input_file
Data, 11
size
Data, 13
skip
stbi_io_callbacks, 27
Slice, 25
slice_xz, 25
slice_yz, 26
slice/Slice.cpp, 45
slice/Slice.h, 45
slice_xz
Slice, 25
slice_yz
Slice, 26
stb/stb_image.h, 46, 57
stb/stb_image_write.h, 151, 156
stb_image.h
stbi_convert_iphone_png_to_rgb, 49
stbi_convert_iphone_png_to_rgb_thread, 49
STBI_default, 48
stbi_failure_reason, 49
STBI_grey, 48
STBI_grey_alpha, 48
stbi_hdr_to_ldr_gamma, 49
stbi_hdr_to_ldr_scale, 49
stbi_image_free, 49
stbi_info, 49
stbi_info_from_callbacks, 50
stbi_info_from_file, 50
stbi_info_from_memory, 50
stbi_is_16_bit, 50
stbi_is_16_bit_from_callbacks, 50

stbi_is_16_bit_from_file, 51
stbi_is_16_bit_from_memory, 51
stbi_is_hdr, 51
stbi_is_hdr_from_callbacks, 51
stbi_is_hdr_from_file, 51
stbi_is_hdr_from_memory, 51
stbi_ldr_to_hdr_gamma, 52
stbi_ldr_to_hdr_scale, 52
stbi_load, 52
stbi_load_16, 52
stbi_load_16_from_callbacks, 52
stbi_load_16_from_memory, 53
stbi_load_from_callbacks, 53
stbi_load_from_file, 53
stbi_load_from_file_16, 53
stbi_load_from_memory, 54
stbi_load_gif_from_memory, 54
stbi_loadf, 54
stbi_loadf_from_callbacks, 54
stbi_loadf_from_file, 55
stbi_loadf_from_memory, 55
STBI_rgb, 48
STBI_rgb_alpha, 48
stbi_set_flip_vertically_on_load, 55
stbi_set_flip_vertically_on_load_thread, 55
stbi_set_unpremultiply_on_load, 55
stbi_set_unpremultiply_on_load_thread, 56
stbi_uc, 48
stbi_us, 48
STBI_VERSION, 48
stbi_zlib_decode_buffer, 56
stbi_zlib_decode_malloc, 56
stbi_zlib_decode_malloc_guesssize, 56
stbi_zlib_decode_malloc_guesssize_headerflag, 56
stbi_zlib_decode_noheader_buffer, 56
stbi_zlib_decode_noheader_malloc, 57
STBIDEF, 48
STB_IMAGE_IMPLEMENTATION
Data.cpp, 37
minimal.cpp, 43
test.cpp, 178
stb_image_write.h
stbi_flip_vertically_on_write, 152
stbi_write_bmp, 153
stbi_write_bmp_to_func, 153
stbi_write_force_png_filter, 155
stbi_write_func, 152
stbi_write_hdr, 153
stbi_write_hdr_to_func, 153
stbi_write_jpg, 153
stbi_write_jpg_to_func, 154
stbi_write_png, 154
stbi_write_png_compression_level, 155
stbi_write_png_to_func, 154
stbi_write_tga, 154
stbi_write_tga_to_func, 155
stbi_write_tga_with_rle, 155
STBIWDEF, 152
STB_IMAGE_WRITE_IMPLEMENTATION
Data.cpp, 37
minimal.cpp, 43
test.cpp, 178
stbi_convert_iphone_png_to_rgb
stb_image.h, 49
stbi_convert_iphone_png_to_rgb_thread
stb_image.h, 49
STBI_default
stb_image.h, 48
stbi_failure_reason
stb_image.h, 49
stbi_flip_vertically_on_write
stb_image_write.h, 152
STBI_grey
stb_image.h, 48
STBI_grey_alpha
stb_image.h, 48
stbi_hdr_to_ldr_gamma
stb_image.h, 49
stbi_hdr_to_ldr_scale
stb_image.h, 49
stbi_image_free
stb_image.h, 49
stbi_info
stb_image.h, 49
stbi_info_from_callbacks
stb_image.h, 50
stbi_info_from_file
stb_image.h, 50
stbi_info_from_memory
stb_image.h, 50
stbi_io_callbacks, 26
eof, 26
read, 26
skip, 27
stbi_is_16_bit
stb_image.h, 50
stbi_is_16_bit_from_callbacks
stb_image.h, 50
stbi_is_16_bit_from_file
stb_image.h, 51
stbi_is_16_bit_from_memory
stb_image.h, 51
stbi_is_hdr
stb_image.h, 51
stbi_is_hdr_from_callbacks
stb_image.h, 51
stbi_is_hdr_from_file
stb_image.h, 51
stbi_is_hdr_from_memory
stb_image.h, 51
stbi_ldr_to_hdr_gamma
stb_image.h, 52
stbi_ldr_to_hdr_scale
stb_image.h, 52
stbi_load

stb_image.h, 52
stbi_load_16
stb_image.h, 52
stbi_load_16_from_callbacks
stb_image.h, 52
stbi_load_16_from_memory
stb_image.h, 53
stbi_load_from_callbacks
stb_image.h, 53
stbi_load_from_file
stb_image.h, 53
stbi_load_from_file_16
stb_image.h, 53
stbi_load_from_memory
stb_image.h, 54
stbi_load_gif_from_memory
stb_image.h, 54
stbi_loadf
stb_image.h, 54
stbi_loadf_from_callbacks
stb_image.h, 54
stbi_loadf_from_file
stb_image.h, 55
stbi_loadf_from_memory
stb_image.h, 55
STBI_rgb
stb_image.h, 48
STBI_rgb_alpha
stb_image.h, 48
stbi_set_flip_vertically_on_load
stb_image.h, 55
stbi_set_flip_vertically_on_load_thread
stb_image.h, 55
stbi_set_unpremultiply_on_load
stb_image.h, 55
stbi_set_unpremultiply_on_load_thread
stb_image.h, 56
stbi_uc
stb_image.h, 48
stbi_us
stb_image.h, 48
STBI_VERSION
stb_image.h, 48
stbi_write_bmp
stb_image_write.h, 153
stbi_write_bmp_to_func
stb_image_write.h, 153
stbi_write_force_png_filter
stb_image_write.h, 155
stbi_write_func
stb_image_write.h, 152
stbi_write_hdr
stb_image_write.h, 153
stbi_write_hdr_to_func
stb_image_write.h, 153
stbi_write_jpg
stb_image_write.h, 153
stbi_write_jpg_to_func
stb_image_write.h, 154
stbi_write_png
stb_image_write.h, 154
stbi_write_png_compression_level
stb_image_write.h, 155
stbi_write_png_to_func
stb_image_write.h, 154
stbi_write_tga
stb_image_write.h, 154
stbi_write_tga_to_func
stb_image_write.h, 155
stbi_write_tga_with_rle
stb_image_write.h, 155
stbi_zlib_decode_buffer
stb_image.h, 56
stbi_zlib_decode_malloc
stb_image.h, 56
stbi_zlib_decode_malloc_guesssize
stb_image.h, 56
stbi_zlib_decode_malloc_guesssize_headerflag
stb_image.h, 56
stbi_zlib_decode_noheader_buffer
stb_image.h, 56
stbi_zlib_decode_noheader_malloc
stb_image.h, 57
STBIDEF
stb_image.h, 48
STBIWDEF
stb_image_write.h, 152

test.cpp, 177
main, 178
OUTPUT_DIR, 177
STB_IMAGE_IMPLEMENTATION, 178
STB_IMAGE_WRITE_IMPLEMENTATION, 178
test_3D_GaussianFilter
Unittest, 28
test_3D_median
Unittest, 28
test_adjust_brightness
Unittest, 28
test_color_balance
Unittest, 28
test_edge_detection
Unittest, 29
test_GaussianFilter
Unittest, 29
test_GaussianFilter3d
Unittest, 29
test_histogram_equalization
Unittest, 29
test_mean_AIP
Unittest, 29
test_median_AIP
Unittest, 30
test_median_filter
Unittest, 30
test_MinIP
Unittest, 30

test_MIP
    Unittest, 31
test_to_grayscale
    Unittest, 31
to_grayscale
    ImageFilter, 21

Unittest, 27
    run_all, 28
    test_3D_GaussianFilter, 28
    test_3D_median, 28
    test_adjust_brightness, 28
    test_color_balance, 28
    test_edge_detection, 29
    test_GaussianFilter, 29
    test_GaussianFilter3d, 29
    test_histogram_equalization, 29
    test_mean_AIP, 29
    test_median_AIP, 30
    test_median_filter, 30
    test_MinIP, 30
    test_MIP, 31
    test_to_grayscale, 31
unittest/Unittest.cpp, 178
unittest/Unittest.h, 178, 179
update
    Data, 12

Volume, 32
    ∼Volume, 34
    dir_name, 34
    get_channels, 34
    get_x, 34
    get_y, 34
    get_z, 34
    Volume, 33
volume/Volume.cpp, 179
volume/Volume.h, 179, 180
VolumeFilter, 35
    ∼VolumeFilter, 35
    GaussianFilter3d, 36
    median3D_filter, 36

w
    Data, 14
write_file
    Data, 12