

ProyectoFinal

Generated by Doxygen 1.9.4

1 Class Index	1
1.1 Class List	1
2 File Index	3
2.1 File List	3
3 Class Documentation	5
3.1 _frame Struct Reference	5
3.2 BoneMatrix Struct Reference	5
3.3 Camera Class Reference	6
3.4 Mesh Class Reference	6
3.5 MeshAnim Class Reference	6
3.6 Model Class Reference	7
3.7 ModelAnim Class Reference	7
3.8 Shader Class Reference	8
3.9 stbi_io_callbacks Struct Reference	8
3.10 Texture Struct Reference	8
3.11 TextureLoading Class Reference	9
3.12 Vertex Struct Reference	9
3.13 VertexBoneData Struct Reference	9
4 File Documentation	11
4.1 419048901_Proyecto_Gpo04.cpp File Reference	11
4.1.1 Detailed Description	12
4.1.2 Function Documentation	13
4.1.2.1 main()	13
4.1.3 Variable Documentation	13
4.1.3.1 pointLightPositions	13
4.1.3.2 spotLightDir	13
4.2 Camera.h	14
4.3 Mesh.h	16
4.4 meshAnim.h	18
4.5 Model.h	20
4.6 modelAnim.h	23
4.7 Shader.h	30
4.8 stb_image.h	32
4.9 Texture.h	117
Index	119

Chapter 1

Class Index

1.1 Class List

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

_frame	5
BoneMatrix	5
Camera	6
Mesh	6
MeshAnim	6
Model	7
ModelAnim	7
Shader	8
stbi_io_callbacks	8
Texture	8
TextureLoading	9
Vertex	9
VertexBoneData	9

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

419048901_Proyecto_Gpo04.cpp	
Archivo principal CPP (main program) del proyecto	11
Camera.h	??
Mesh.h	??
meshAnim.h	??
Model.h	??
modelAnim.h	??
Shader.h	??
stb_image.h	??
Texture.h	??

Chapter 3

Class Documentation

3.1 `_frame` Struct Reference

Public Attributes

- float **posX**
- float **posY**
- float **posZ**
- float **incX**
- float **incY**
- float **incZ**
- float **rotRodlZq**
- float **rotInc**

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

- MainPrueba.cpp

3.2 `BoneMatrix` Struct Reference

Public Attributes

- aiMatrix4x4 **offset_matrix**
- aiMatrix4x4 **final_world_transform**

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

- meshAnim.h

3.3 Camera Class Reference

Public Member Functions

- **Camera** (glm::vec3 position=glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up=glm::vec3(0.0f, 1.0f, 0.0f), GLfloat yaw=YAW, GLfloat pitch=PITCH)
- **Camera** (GLfloat posX, GLfloat posY, GLfloat posZ, GLfloat upX, GLfloat upY, GLfloat upZ, GLfloat yaw, GLfloat pitch)
- glm::mat4 **GetViewMatrix** ()
- void **Recorrido** (GLfloat xOffset)
- void **MovimientoAutomatico** (GLfloat velocidad)
- void **ProcessKeyboard** (Camera_Movement direction, GLfloat deltaTime)
- void **ProcessMouseMovement** (GLfloat xOffset, GLfloat yOffset, GLboolean constrainPitch=true)
- void **ProcessMouseScroll** (GLfloat yOffset)
- GLfloat **GetZoom** ()
- glm::vec3 **GetPosition** ()
- glm::vec3 **GetFront** ()

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

- Camera.h

3.4 Mesh Class Reference

Public Member Functions

- **Mesh** (vector< [Vertex](#) > vertices, vector< unsigned int > indices, vector< [Texture](#) > textures)
- void **Draw** ([Shader](#) shader)

Public Attributes

- vector< [Vertex](#) > **vertices**
- vector< unsigned int > **indices**
- vector< [Texture](#) > **textures**
- unsigned int **VAO**

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

- Mesh.h

3.5 MeshAnim Class Reference

Public Member Functions

- **MeshAnim** (vector< [Vertex](#) > vertices, vector< unsigned int > indices, vector< [Texture](#) > textures)
- **MeshAnim** (vector< [Vertex](#) > vertices, vector< unsigned int > indices, vector< [Texture](#) > textures, vector< [VertexBoneData](#) > bone_id_weights)
- void **Draw** ([Shader](#) shader)

Public Attributes

- vector< [Vertex](#) > **vertices**
- vector< unsigned int > **indices**
- vector< [Texture](#) > **textures**
- vector< [VertexBoneData](#) > **bones_id_weights_for_each_vertex**
- unsigned int **VAO**

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

- meshAnim.h

3.6 Model Class Reference

Public Member Functions

- **Model** (string const &path, bool gamma=false)
- void **Draw** ([Shader](#) shader)

Public Attributes

- vector< [Texture](#) > **textures_loaded**
- vector< [Mesh](#) > **meshes**
- string **directory**
- bool **gammaCorrection**

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

- Model.h

3.7 ModelAnim Class Reference

Public Member Functions

- **ModelAnim** (string const &path, bool gamma=false)
- void **initShaders** (GLuint shader_program)
- void **Draw** ([Shader](#) shader)

Public Attributes

- vector< [Texture](#) > **textures_loaded**
- vector< [MeshAnim](#) > **meshes**
- string **directory**
- bool **gammaCorrection**
- Assimp::Importer **importer**
- const aiScene * **scene**
- map< string, uint > **m_bone_mapping**
- uint **m_num_bones** = 0
- vector< [BoneMatrix](#) > **m_bone_matrices**
- aiMatrix4x4 **m_global_inverse_transform**
- GLuint **m_bone_location** [MAX_BONES]
- float **ticks_per_second** = 0.0f

Static Public Attributes

- static const uint **MAX_BONES** = 100

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

- modelAnim.h

3.8 Shader Class Reference

Public Member Functions

- **Shader** (const GLchar *vertexPath, const GLchar *fragmentPath)
- void **Use** ()
- GLuint **getColorLocation** ()

Public Attributes

- GLuint **Program**
- GLuint **uniformColor**

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

- Shader.h

3.9 stbi_io_callbacks Struct Reference

Public Attributes

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

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

- stb_image.h

3.10 Texture Struct Reference

Public Attributes

- unsigned int **id**
- string **type**
- string **path**

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

- Mesh.h

3.11 TextureLoading Class Reference

Static Public Member Functions

- static GLuint **LoadTexture** (GLchar *path)
- static GLuint **LoadCubemap** (vector< const GLchar * > faces)

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

- Texture.h

3.12 Vertex Struct Reference

Public Attributes

- glm::vec3 **Position**
- glm::vec3 **Normal**
- glm::vec2 **TexCoords**
- glm::vec3 **Tangent**
- glm::vec3 **Bitangent**

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

- Mesh.h

3.13 VertexBoneData Struct Reference

Public Member Functions

- void **addBoneData** (uint bone_id, float weight)

Public Attributes

- uint **ids** [NUM_BONES_PER_VEREX]
- float **weights** [NUM_BONES_PER_VEREX]

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

- meshAnim.h

Chapter 4

File Documentation

4.1 419048901_Proyecto_Gpo04.cpp File Reference

Archivo principal CPP (main program) del proyecto.

```
#include <iostream>
#include <cmath>
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include "stb_image.h"
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include "SOIL2/SOIL2.h"
#include "Shader.h"
#include "Camera.h"
#include "Model.h"
#include "Texture.h"
#include "modelAnim.h"
```

Functions

- void **KeyCallback** (GLFWwindow *window, int key, int scancode, int action, int mode)
- void **MouseButtonCallback** (GLFWwindow *window, double xPos, double yPos)
- void **DoMovement** ()

Modifica posiciones de Camara respecto a Entradas de Usuario.

- void **animacion** ()

Realiza animaciones de objetos, modificando las variables para operaciones basicas.

- glm::vec3 **lightPos** (0.0f, 0.0f, 0.0f)
- glm::vec3 **PosIni** (-16.0f, 1.0f, -70.0f)
- glm::vec3 **lightDirection** (0.0f, -1.0f, -1.0f)
- glm::vec3 **PosIniCar** (80.0f, 0.0f, 14.0f)
- glm::vec3 **PosIniPerson** (-16.0f, 0.0f, -70.0f)
- int **main** ()

Funcion del programa principal.

Variables

- const GLuint **WIDTH** = 800
- const GLuint **HEIGHT** = 600
- int **SCREEN_WIDTH**
- int **SCREEN_HEIGHT**
- Camera **camera** (glm::vec3(0.0f, 10.0f, 25.0f))
- GLfloat **lastX** = WIDTH / 2.0
- GLfloat **lastY** = HEIGHT / 2.0
- bool **keys** [1024]
- bool **firstMouse** = true
- bool **active**
- bool **encendido** = false
- glm::vec3 **pointLightPositions** []
- glm::vec3 **spotLightPosition** = glm::vec3(0.0f, 19.0f, 0.0f)
- int **dir** = 0
- glm::vec3 **spotLightDir** []
- float **vertices** []
- GLfloat **skyboxVertices** []
- glm::vec3 **Light1** = glm::vec3(0)
- glm::vec3 **Light2** = glm::vec3(0)
- glm::vec3 **Light3** = glm::vec3(0)
- glm::vec3 **Light4** = glm::vec3(0)
- float **rotDoor** = 0.0f
- bool **actionDoor** = false
- bool **openDoor** = false

Variables Animaci Puerta.

- float **rotCam** = 0.0
- bool **CamDerecha** = false

Variables Animaci Camara Seguridad.

- float **movKitX** = 0.0
- float **movKitZ** = 0.0
- float **rotKit** = 0.0
- bool **circuito** = false
- bool **recorrido1** = true
- bool **recorrido2** = false
- bool **recorrido3** = false
- bool **recorrido4** = false
- bool **recorrido5** = false
- bool **recorrido6** = false
- bool **recorrido7** = false
- bool **recorrido8** = false
- GLfloat **deltaTime** = 0.0f
- GLfloat **lastFrame** = 0.0f

4.1.1 Detailed Description

Archivo principal CPP (main program) del proyecto.

Author

NumCuenta: 419048901

Date

11/05/2022

4.1.2 Function Documentation

4.1.2.1 main()

```
int main ( )
```

Funcion del programa principal.

Returns

Devuelve 0 de programa exitoso

4.1.3 Variable Documentation

4.1.3.1 pointLightPositions

```
glm::vec3 pointLightPositions[]
```

Initial value:

```
= {  
    glm::vec3(0.0f, 19.0f, 0.0f)  
}
```

4.1.3.2 spotLightDir

```
glm::vec3 spotLightDir[]
```

Initial value:

```
= {  
    glm::vec3(0.0f,-1.0f, 0.0f),  
    glm::vec3(1.0f,0.0f, 0.0f),  
    glm::vec3(0.0f,0.0f, -1.0f),  
    glm::vec3(-1.0f,0.0f, 0.0f),  
    glm::vec3(0.0f,0.0f, 1.0f),  
    glm::vec3(0.0f,1.0f, 0.0f),  
    glm::vec3(0.0f,-1.0f, 0.0f)  
}
```

4.2 Camera.h

```

1 #pragma once
2
3 // Std. Includes
4 #include <vector>
5
6 // GL Includes
7 #define GLEW_STATIC
8 #include <GL/glew.h>
9
10 #include <glm/glm.hpp>
11 #include <glm/gtc/matrix_transform.hpp>
12
13 // Defines several possible options for camera movement. Used as abstraction to stay away from
    window-system specific input methods
14 enum Camera_Movement
15 {
16     FORWARD,
17     BACKWARD,
18     LEFT,
19     RIGHT
20 };
21
22 // Default camera values
23 const GLfloat YAW = -90.0f;
24 const GLfloat PITCH = 0.0f;
25 const GLfloat SPEED = 10.0f;
26 const GLfloat SENSITIVITY = 0.25f;
27 const GLfloat ZOOM = 45.0f;
28
29 // An abstract camera class that processes input and calculates the corresponding Euler Angles, Vectors
    and Matrices for use in OpenGL
30 class Camera
31 {
32 public:
33     // Constructor with vectors
34     Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f),
        GLfloat yaw = YAW, GLfloat pitch = PITCH) : front(glm::vec3(0.0f, 0.0f, -1.0f)),
        movementSpeed(SPEED), mouseSensitivity(SENSITIVITY), zoom(ZOOM)
35     {
36         this->position = position;
37         this->worldUp = up;
38         this->yaw = yaw;
39         this->pitch = pitch;
40         this->updateCameraVectors();
41     }
42
43     // Constructor with scalar values
44     Camera(GLfloat posX, GLfloat posY, GLfloat posZ, GLfloat upX, GLfloat upY, GLfloat upZ, GLfloat yaw,
        GLfloat pitch) : front(glm::vec3(0.0f, 0.0f, -1.0f)), movementSpeed(SPEED),
        mouseSensitivity(SENSITIVITY), zoom(ZOOM)
45     {
46         this->position = glm::vec3(posX, posY, posZ);
47         this->worldUp = glm::vec3(upX, upY, upZ);
48         this->yaw = yaw;
49         this->pitch = pitch;
50         this->updateCameraVectors();
51     }
52
53     // Returns the view matrix calculated using Euler Angles and the LookAt Matrix
54     glm::mat4 GetViewMatrix()
55     {
56         return glm::lookAt(this->position, this->position + this->front, this->up);
57     }
58
59     void Recorrido(GLfloat xOffset) //Modifica la rotacion cambiando el ulo
60     {
61         this->yaw = xOffset;
62         this->updateCameraVectors();
63     }
64
65     void MovimientoAutomatico(GLfloat velocidad) //Realiza un movimiento automatico hacia adelante
66     {
67         this->position += this->front * velocidad;
68     }
69
70
71     // Processes input received from any keyboard-like input system. Accepts input parameter in the form
    of camera defined ENUM (to abstract it from windowing systems)
72     void ProcessKeyboard(Camera_Movement direction, GLfloat deltaTime)
73     {
74         GLfloat velocity = this->movementSpeed * deltaTime;
75
76         if (direction == FORWARD)
77         {
78             this->position += this->front * velocity;

```

```

79     }
80
81     if (direction == BACKWARD)
82     {
83         this->position -= this->front * velocity;
84     }
85
86     if (direction == LEFT)
87     {
88         this->position -= this->right * velocity;
89     }
90
91     if (direction == RIGHT)
92     {
93         this->position += this->right * velocity;
94     }
95 }
96
97 // Processes input received from a mouse input system. Expects the offset value in both the x and y
98 // direction.
99 void ProcessMouseMove(GLfloat xOffset, GLfloat yOffset, GLboolean constrainPitch = true)
100 {
101     xOffset *= this->mouseSensitivity;
102     yOffset *= this->mouseSensitivity;
103
104     this->yaw += xOffset;
105     this->pitch += yOffset;
106
107     // Make sure that when pitch is out of bounds, screen doesn't get flipped
108     if (constrainPitch)
109     {
110         if (this->pitch > 100.0f)
111         {
112             this->pitch = 89.0f;
113         }
114         if (this->pitch < -89.0f)
115         {
116             this->pitch = -89.0f;
117         }
118     }
119
120     // Update Front, Right and Up Vectors using the updated Euler angles
121     this->updateCameraVectors();
122 }
123
124
125 // Processes input received from a mouse scroll-wheel event. Only requires input on the vertical
126 // wheel-axis
127 void ProcessMouseScroll(GLfloat yOffset)
128 {
129 }
130
131 GLfloat GetZoom()
132 {
133     return this->zoom;
134 }
135
136 glm::vec3 GetPosition()
137 {
138     return this->position;
139 }
140
141 glm::vec3 GetFront()
142 {
143     return this->front;
144 }
145
146 private:
147     // Camera Attributes
148     glm::vec3 position;
149     glm::vec3 front;
150     glm::vec3 up;
151     glm::vec3 right;
152     glm::vec3 worldUp;
153
154     // Euler Angles
155     GLfloat yaw;
156     GLfloat pitch;
157
158     // Camera options
159     GLfloat movementSpeed;
160     GLfloat mouseSensitivity;
161     GLfloat zoom;
162
163     // Calculates the front vector from the Camera's (updated) Euler Angles

```

```

164     void updateCameraVectors()
165     {
166         // Calculate the new Front vector
167         glm::vec3 front;
168         front.x = cos(glm::radians(this->yaw)) * cos(glm::radians(this->pitch));
169         front.y = sin(glm::radians(this->pitch));
170         front.z = sin(glm::radians(this->yaw)) * cos(glm::radians(this->pitch));
171         this->front = glm::normalize(front);
172         // Also re-calculate the Right and Up vector
173         this->right = glm::normalize(glm::cross(this->front, this->worldUp)); // Normalize the vectors,
        because their length gets closer to 0 the more you look up or down which results in slower movement.
174         this->up = glm::normalize(glm::cross(this->right, this->front));
175     }
176 };

```

4.3 Mesh.h

```

1  #ifndef MESH_H
2  #define MESH_H
3
4  // #include "glad.h" // holds all OpenGL type declarations
5
6  #include <glm/glm.hpp>
7  #include <glm/gtc/matrix_transform.hpp>
8
9  #include "shader.h"
10
11 #include <string>
12 #include <fstream>
13 #include <sstream>
14 #include <iostream>
15 #include <vector>
16 using namespace std;
17
18 struct Vertex {
19     // position
20     glm::vec3 Position;
21     // normal
22     glm::vec3 Normal;
23     // texCoords
24     glm::vec2 TexCoords;
25     // tangent
26     glm::vec3 Tangent;
27     // bitangent
28     glm::vec3 Bitangent;
29 };
30
31 struct Texture {
32     unsigned int id;
33     string type;
34     string path;
35 };
36
37 class Mesh {
38 public:
39     /* Mesh Data */
40     vector<Vertex> vertices;
41     vector<unsigned int> indices;
42     vector<Texture> textures;
43     unsigned int VAO;
44
45     /* Functions */
46     // constructor
47     Mesh(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures)
48     {
49         this->vertices = vertices;
50         this->indices = indices;
51         this->textures = textures;
52
53         // now that we have all the required data, set the vertex buffers and its attribute pointers.
54         setupMesh();
55     }
56
57     // render the mesh
58     void Draw(Shader shader)
59     {
60         // bind appropriate textures
61         unsigned int diffuseNr = 1;
62         unsigned int specularNr = 1;
63         unsigned int normalNr = 1;
64         unsigned int heightNr = 1;
65         for(unsigned int i = 0; i < textures.size(); i++)
66         {

```

```

67         glActiveTexture(GL_TEXTURE0 + i); // active proper texture unit before binding
68         // retrieve texture number (the N in diffuse_textureN)
69         string number;
70         string name = textures[i].type;
71         if(name == "texture_diffuse")
72             number = std::to_string(diffuseNr++);
73         else if(name == "texture_specular")
74             number = std::to_string(specularNr++); // transfer unsigned int to stream
75         else if(name == "texture_normal")
76             number = std::to_string(normalNr++); // transfer unsigned int to stream
77         else if(name == "texture_height")
78             number = std::to_string(heightNr++); // transfer unsigned int to stream
79
80         // now set the sampler to the correct texture unit
81         glUniform1i(glGetUniformLocation(shader.Program, (name + number).c_str()), i); // AQUI ES
DONDE SE ASIGNAN LOS UNIFORM A LOS SHADERS AHHHHHHHHHHHHH
82         // and finally bind the texture
83         glBindTexture(GL_TEXTURE_2D, textures[i].id);
84     }
85
86     // draw mesh
87     glBindVertexArray(VAO);
88     glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
89     glBindVertexArray(0);
90
91     // always good practice to set everything back to defaults once configured.
92     glActiveTexture(GL_TEXTURE0);
93 }
94
95 private:
96     /* Render data */
97     unsigned int VBO, EBO;
98
99     /* Functions */
100    // initializes all the buffer objects/arrays
101    void setupMesh()
102    {
103        // create buffers/arrays
104        glGenVertexArrays(1, &VAO);
105        glGenBuffers(1, &VBO);
106        glGenBuffers(1, &EBO);
107
108        glBindVertexArray(VAO);
109        // load data into vertex buffers
110        glBindBuffer(GL_ARRAY_BUFFER, VBO);
111        // A great thing about structs is that their memory layout is sequential for all its items.
112        // The effect is that we can simply pass a pointer to the struct and it translates perfectly to
a glm::vec3/2 array which
113        // again translates to 3/2 floats which translates to a byte array.
114        glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);
115
116        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
117        glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), &indices[0],
GL_STATIC_DRAW);
118
119        // set the vertex attribute pointers
120        // vertex Positions
121        glEnableVertexAttribArray(0);
122        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
123        // vertex normals
124        glEnableVertexAttribArray(1);
125        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
Normal));
126        // vertex texture coords
127        glEnableVertexAttribArray(2);
128        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
TexCoords));
129        // vertex tangent
130        glEnableVertexAttribArray(3);
131        glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
Tangent));
132        // vertex bitangent
133        glEnableVertexAttribArray(4);
134        glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
Bitangent));
135
136        glBindVertexArray(0);
137    }
138 };
139 #endif
140

```

4.4 meshAnim.h

```

1  #ifndef MESH_ANIM_H
2  #define MESH_ANIM_H
3
4  // #include <glad.h> // holds all OpenGL type declarations
5
6  #include <glm/glm.hpp>
7  #include <glm/gtc/matrix_transform.hpp>
8
9  #include "shader.h"
10 #include "mesh.h"
11
12 #include <string>
13 #include <fstream>
14 #include <sstream>
15 #include <iostream>
16 #include <vector>
17 using namespace std;
18
19 typedef unsigned int uint;
20 #define NUM_BONES_PER_VERTEX 4
21
22 struct BoneMatrix
23 {
24     aiMatrix4x4 offset_matrix;
25     aiMatrix4x4 final_world_transform;
26 };
27
28 struct VertexBoneData
29 {
30     uint ids[NUM_BONES_PER_VERTEX]; // we have 4 bone ids for EACH vertex & 4 weights for EACH vertex
31     float weights[NUM_BONES_PER_VERTEX];
32
33     VertexBoneData()
34     {
35         memset(ids, 0, sizeof(ids)); // init all values in array = 0
36         memset(weights, 0, sizeof(weights));
37     }
38
39     void addBoneData(uint bone_id, float weight)
40     {
41         for (uint i = 0; i < NUM_BONES_PER_VERTEX; i++)
42         {
43             if (weights[i] == 0.0)
44             {
45                 ids[i] = bone_id;
46                 weights[i] = weight;
47                 return;
48             }
49         }
50     }
51 };
52
53 class MeshAnim {
54 public:
55     /* Mesh Data */
56     vector<Vertex> vertices;
57     vector<unsigned int> indices;
58     vector<Texture> textures;
59     vector<VertexBoneData> bones_id_weights_for_each_vertex;
60     unsigned int VAO;
61
62     /* Functions */
63     // constructor
64     MeshAnim(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures)
65     {
66         this->vertices = vertices;
67         this->indices = indices;
68         this->textures = textures;
69
70         // now that we have all the required data, set the vertex buffers and its attribute pointers.
71         setupMesh();
72     }
73
74     MeshAnim(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures,
75             vector<VertexBoneData> bone_id_weights)
76     {
77         this->vertices = vertices;
78         this->indices = indices;
79         this->textures = textures;
80         bones_id_weights_for_each_vertex = bone_id_weights;
81
82         // now that we have all the required data, set the vertex buffers and its attribute pointers.
83         setupMesh();
84     }

```

```

85     // render the mesh
86     void Draw(Shader shader)
87     {
88         // bind appropriate textures
89         unsigned int diffuseNr  = 1;
90         unsigned int specularNr = 1;
91         unsigned int normalNr   = 1;
92         unsigned int heightNr   = 1;
93         for(unsigned int i = 0; i < textures.size(); i++)
94         {
95             glActiveTexture(GL_TEXTURE0 + i); // active proper texture unit before binding
96             // retrieve texture number (the N in diffuse_textureN)
97             string number;
98             string name = textures[i].type;
99             if(name == "texture_diffuse")
100                 number = std::to_string(diffuseNr++);
101             else if(name == "texture_specular")
102                 number = std::to_string(specularNr++); // transfer unsigned int to stream
103             else if(name == "texture_normal")
104                 number = std::to_string(normalNr++); // transfer unsigned int to stream
105             else if(name == "texture_height")
106                 number = std::to_string(heightNr++); // transfer unsigned int to stream
107
108             // now set the sampler to the correct texture unit
109             glUniform1i(glGetUniformLocation(shader.Program, (name + number).c_str()), i);    // AQUI ES
DONDE SE ASIGNAN LOS UNIFORM A LOS SHADERS AHHHHHHHHHHHHH
110             // and finally bind the texture
111             glBindTexture(GL_TEXTURE_2D, textures[i].id);
112         }
113
114         // draw mesh
115         glBindVertexArray(VAO);
116         glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
117         glBindVertexArray(0);
118
119         // always good practice to set everything back to defaults once configured.
120         glActiveTexture(GL_TEXTURE0);
121     }
122
123 private:
124     /* Render data */
125     unsigned int VBO, EBO, VBO_bones;
126
127     /* Functions */
128     // initializes all the buffer objects/arrays
129     void setupMesh()
130     {
131         // create buffers/arrays
132         glGenVertexArrays(1, &VAO);
133         glGenBuffers(1, &VBO);
134         glGenBuffers(1, &EBO);
135         glGenBuffers(1, &VBO_bones);
136
137         glBindVertexArray(VAO);
138         // load data into vertex buffers
139         glBindBuffer(GL_ARRAY_BUFFER, VBO);
140         // A great thing about structs is that their memory layout is sequential for all its items.
141         // The effect is that we can simply pass a pointer to the struct and it translates perfectly to
a glm::vec3/2 array which
142         // again translates to 3/2 floats which translates to a byte array.
143         glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);
144
145         // bones
146         glBindBuffer(GL_ARRAY_BUFFER, VBO_bones);
147         glBufferData(GL_ARRAY_BUFFER, bones_id_weights_for_each_vertex.size() *
sizeof(bones_id_weights_for_each_vertex[0]), &bones_id_weights_for_each_vertex[0], GL_STATIC_DRAW);
148
149         // indices
150         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
151         glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), &indices[0],
GL_STATIC_DRAW);
152
153         // Se liga primero el buffer de los vertices
154         glBindBuffer(GL_ARRAY_BUFFER, VBO);
155         // set the vertex attribute pointers
156         // vertex Positions
157         glEnableVertexAttribArray(0);
158         glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
159         // vertex normals
160         glEnableVertexAttribArray(1);
161         glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
Normal));
162         // vertex texture coords
163         glEnableVertexAttribArray(2);
164         glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
TexCoords));
165         // vertex tangent

```

```

166         glEnableVertexAttribArray(3);
167         glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
Tangent));
168         // vertex bitangent
169         glEnableVertexAttribArray(4);
170         glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
Bitangent));
171
172         // Se liga el buffer de los bones
173         glBindBuffer(GL_ARRAY_BUFFER, VBO_bones);
174         // set the bones attribute pointers
175         glEnableVertexAttribArray(5);
176         glVertexAttribIPointer(5, 4, GL_INT, sizeof(VertexBoneData), (void*)0);
177         glEnableVertexAttribArray(6);
178         glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(VertexBoneData),
(void*)offsetof(VertexBoneData, weights));
179
180         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
181         glBindVertexArray(0);
182     }
183 };
184 #endif
185

```

4.5 Model.h

```

1 #ifndef MODEL_H
2 #define MODEL_H
3
4 // #include "glad.h"
5
6 #include <glm/glm.hpp>
7 #include <glm/gtc/matrix_transform.hpp>
8
9 #include <assimp/Importer.hpp>
10 #include <assimp/scene.h>
11 #include <assimp/postprocess.h>
12
13 #include "mesh.h"
14 #include "shader.h"
15
16 #include <string>
17 #include <fstream>
18 #include <sstream>
19 #include <iostream>
20 #include <map>
21 #include <vector>
22 using namespace std;
23
24 unsigned int TextureFromFile(const char *path, const string &directory, bool gamma = false);
25
26 class Model
27 {
28 public:
29     /* Model Data */
30     vector<Texture> textures_loaded; // stores all the textures loaded so far, optimization to make
sure textures aren't loaded more than once.
31     vector<Mesh> meshes;
32     string directory;
33     bool gammaCorrection;
34
35     /* Functions */
36     // constructor, expects a filepath to a 3D model.
37     Model(string const &path, bool gamma = false) : gammaCorrection(gamma)
38     {
39         loadModel(path);
40     }
41
42     // draws the model, and thus all its meshes
43     void Draw(Shader shader)
44     {
45         for(unsigned int i = 0; i < meshes.size(); i++)
46             meshes[i].Draw(shader);
47     }
48
49 private:
50     /* Functions */
51     // loads a model with supported ASSIMP extensions from file and stores the resulting meshes in the
meshes vector.
52     void loadModel(string const &path)
53     {
54         // read file via ASSIMP
55         Assimp::Importer importer;

```



```

56     const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs |
aiProcess_CalcTangentSpace);
57     // check for errors
58     if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) // if is Not Zero
59     {
60         cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << endl;
61         return;
62     }
63     // retrieve the directory path of the filepath
64     directory = path.substr(0, path.find_last_of('/'));
65
66     // process ASSIMP's root node recursively
67     processNode(scene->mRootNode, scene);
68 }
69
70 // processes a node in a recursive fashion. Processes each individual mesh located at the node and
repeats this process on its children nodes (if any).
71 void processNode(aiNode *node, const aiScene *scene)
72 {
73     // process each mesh located at the current node
74     for(unsigned int i = 0; i < node->mNumMeshes; i++)
75     {
76         // the node object only contains indices to index the actual objects in the scene.
77         // the scene contains all the data, node is just to keep stuff organized (like relations
between nodes).
78         aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
79         meshes.push_back(processMesh(mesh, scene));
80     }
81     // after we've processed all of the meshes (if any) we then recursively process each of the
children nodes
82     for(unsigned int i = 0; i < node->mNumChildren; i++)
83     {
84         processNode(node->mChildren[i], scene);
85     }
86 }
87
88 Mesh processMesh(aiMesh *mesh, const aiScene *scene)
89 {
90     {
91         // data to fill
92         vector<Vertex> vertices;
93         vector<unsigned int> indices;
94         vector<Texture> textures;
95
96         // Walk through each of the mesh's vertices
97         for(unsigned int i = 0; i < mesh->mNumVertices; i++)
98         {
99             Vertex vertex;
100             glm::vec3 vector; // we declare a placeholder vector since assimp uses its own vector class
that doesn't directly convert to glm's vec3 class so we transfer the data to this placeholder
glm::vec3 first.
101             // positions
102             vector.x = mesh->mVertices[i].x;
103             vector.y = mesh->mVertices[i].y;
104             vector.z = mesh->mVertices[i].z;
105             vertex.Position = vector;
106             // normals
107             vector.x = mesh->mNormals[i].x;
108             vector.y = mesh->mNormals[i].y;
109             vector.z = mesh->mNormals[i].z;
110             vertex.Normal = vector;
111             // texture coordinates
112             if(mesh->mTextureCoords[0]) // does the mesh contain texture coordinates?
113             {
114                 glm::vec2 vec;
115                 // a vertex can contain up to 8 different texture coordinates. We thus make the
assumption that we won't
116                 // use models where a vertex can have multiple texture coordinates so we always take the
first set (0).
117                 vec.x = mesh->mTextureCoords[0][i].x;
118                 vec.y = mesh->mTextureCoords[0][i].y;
119                 vertex.TexCoords = vec;
120             }
121             else
122                 vertex.TexCoords = glm::vec2(0.0f, 0.0f);
123             // tangent
124             /* vector.x = mesh->mTangents[i].x;
125             vector.y = mesh->mTangents[i].y;
126             vector.z = mesh->mTangents[i].z;
127             vertex.Tangent = vector;*/
128             // bitangent
129             /* vector.x = mesh->mBitangents[i].x;
130             vector.y = mesh->mBitangents[i].y;
131             vector.z = mesh->mBitangents[i].z;*/
132             vertex.Bitangent = vector;
133             vertices.push_back(vertex);
134         }

```

```

135         // now walk through each of the mesh's faces (a face is a mesh its triangle) and retrieve the
corresponding vertex indices.
136         for(unsigned int i = 0; i < mesh->mNumFaces; i++)
137         {
138             aiFace face = mesh->mFaces[i];
139             // retrieve all indices of the face and store them in the indices vector
140             for(unsigned int j = 0; j < face.mNumIndices; j++)
141                 indices.push_back(face.mIndices[j]);
142         }
143         // process materials
144         aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
145         // we assume a convention for sampler names in the shaders. Each diffuse texture should be named
146         // as 'texture_diffuseN' where N is a sequential number ranging from 1 to MAX_SAMPLER_NUMBER.
147         // Same applies to other texture as the following list summarizes:
148         // diffuse: texture_diffuseN
149         // specular: texture_specularN
150         // normal: texture_normalN
151
152         // 1. diffuse maps
153         vector<Texture> diffuseMaps = loadMaterialTextures(material, aiTextureType_DIFFUSE,
"texture_diffuse");
154         textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());
155         // 2. specular maps
156         vector<Texture> specularMaps = loadMaterialTextures(material, aiTextureType_SPECULAR,
"texture_specular");
157         textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());
158         // 3. normal maps
159         std::vector<Texture> normalMaps = loadMaterialTextures(material, aiTextureType_HEIGHT,
"texture_normal");
160         textures.insert(textures.end(), normalMaps.begin(), normalMaps.end());
161         // 4. height maps
162         std::vector<Texture> heightMaps = loadMaterialTextures(material, aiTextureType_AMBIENT,
"texture_height");
163         textures.insert(textures.end(), heightMaps.begin(), heightMaps.end());
164
165         // return a mesh object created from the extracted mesh data
166         return Mesh(vertices, indices, textures);
167     }
168
169     // checks all material textures of a given type and loads the textures if they're not loaded yet.
170     // the required info is returned as a Texture struct.
171     vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type, string typeName)
172     {
173         vector<Texture> textures;
174         for(unsigned int i = 0; i < mat->GetTextureCount(type); i++)
175         {
176             aiString str;
177             mat->GetTexture(type, i, &str);
178             // check if texture was loaded before and if so, continue to next iteration: skip loading a
new texture
179             bool skip = false;
180             for(unsigned int j = 0; j < textures_loaded.size(); j++)
181             {
182                 if(std::strcmp(textures_loaded[j].path.data(), str.C_Str()) == 0)
183                 {
184                     textures.push_back(textures_loaded[j]);
185                     skip = true; // a texture with the same filepath has already been loaded, continue
to next one. (optimization)
186                     break;
187                 }
188             }
189             if(!skip)
190             { // if texture hasn't been loaded already, load it
191                 Texture texture;
192                 texture.id = TextureFromFile(str.C_Str(), this->directory);
193                 texture.type = typeName;
194                 texture.path = str.C_Str();
195                 textures.push_back(texture);
196                 textures_loaded.push_back(texture); // store it as texture loaded for entire model, to
ensure we won't unnecesary load duplicate textures.
197             }
198         }
199         return textures;
200     }
201 };
202
203
204 unsigned int TextureFromFile(const char *path, const string &directory, bool gamma)
205 {
206     string filename = string(path);
207     filename = directory + "/" + filename;
208
209     unsigned int textureID;
210     glGenTextures(1, &textureID);
211
212     int width, height, nrComponents;
213     unsigned char *data = stbi_load(filename.c_str(), &width, &height, &nrComponents, 0);

```

```

214     if (data)
215     {
216         GLenum format;
217         if (nrComponents == 1)
218             format = GL_RED;
219         else if (nrComponents == 3)
220             format = GL_RGB;
221         else if (nrComponents == 4)
222             format = GL_RGBA;
223
224         glBindTexture(GL_TEXTURE_2D, textureID);
225         glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
226         glGenerateMipmap(GL_TEXTURE_2D);
227
228         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
229         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
230         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
231         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
232
233         stbi_image_free(data);
234     }
235     else
236     {
237         std::cout << "Texture failed to load at path: " << path << std::endl;
238         stbi_image_free(data);
239     }
240
241     return textureID;
242 }
243 #endif

```

4.6 modelAnim.h

```

1  #ifndef MODEL_ANIM_H
2  #define MODEL_ANIM_H
3
4  // #include "glad.h"
5
6  #include <glm/glm.hpp>
7  #include <glm/gtc/matrix_transform.hpp>
8
9  #include <assimp/Importer.hpp>
10 #include <assimp/scene.h>
11 #include <assimp/postprocess.h>
12
13 #include <GLFW/glfw3.h>
14 #include "meshAnim.h"
15 #include "model.h"
16 #include "shader.h"
17 #include <string>
18 #include <fstream>
19 #include <sstream>
20 #include <iostream>
21 #include <map>
22 #include <vector>
23 using namespace std;
24
25 class ModelAnim
26 {
27 public:
28     /* Model Data */
29     vector<Texture> textures_loaded; // stores all the textures loaded so far, optimization to make
30     sure textures aren't loaded more than once.
31     vector<MeshAnim> meshes;
32     string directory;
33     bool gammaCorrection;
34
35     /* Importacion base */
36     Assimp::Importer importer;
37     const aiScene* scene;
38
39     /* Huesos */
40     static const uint MAX_BONES = 100;
41
42     map<string, uint> m_bone_mapping; // maps a bone name and their index
43     uint m_num_bones = 0;
44     vector<BoneMatrix> m_bone_matrices;
45     aiMatrix4x4 m_global_inverse_transform;
46
47     GLuint m_bone_location[MAX_BONES];
48     float ticks_per_second = 0.0f;
49
50     /* Functions */

```

```

50 // constructor, expects a filepath to a 3D model.
51 ModelAnim(string const &path, bool gamma = false) : gammaCorrection(gamma)
52 {
53     loadModel(path);
54 }
55
56 void initShaders(GLuint shader_program)
57 {
58     for (uint i = 0; i < MAX_BONES; i++) // get location all matrices of bones
59     {
60         string name = "bones[" + to_string(i) + "]; // name like in shader
61         m_bone_location[i] = glGetUniformLocation(shader_program, name.c_str());
62     }
63
64     // rotate head AND AXIS(y_z) about x !!!!! Not be gimbal lock
65     //rotate_head_xz *= glm::quat(cos(glm::radians(-45.0f / 2)), sin(glm::radians(-45.0f / 2)) *
66     glm::vec3(1.0f, 0.0f, 0.0f));
67 }
68
69 // draws the model, and thus all its meshes
70 void Draw(Shader shader)
71 {
72     // Calculo de las animaciones
73     vector<aiMatrix4x4> transforms;
74     boneTransform((double)glfwGetTime(), transforms);
75
76     for (uint i = 0; i < transforms.size(); i++) // move all matrices for actual model position to
77     shader
78     {
79         glUniformMatrix4fv(m_bone_location[i], 1, GL_TRUE, (const GLfloat*)&transforms[i]);
80     }
81
82     for(unsigned int i = 0; i < meshes.size(); i++)
83     meshes[i].Draw(shader);
84 }
85 private:
86
87 /* Functions */
88 // loads a model with supported ASSIMP extensions from file and stores the resulting meshes in the
89 meshes vector.
90 void loadModel(string const &path)
91 {
92     // read file via ASSIMP
93     scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs |
94     aiProcess_CalcTangentSpace);
95     // check for errors
96     if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) // if is Not Zero
97     {
98         cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << endl;
99         return;
100     }
101
102     m_global_inverse_transform = scene->mRootNode->mTransformation;
103     m_global_inverse_transform.Inverse();
104
105     if (scene->mAnimations[0]->mTicksPerSecond != 0.0)
106     {
107         ticks_per_second = scene->mAnimations[0]->mTicksPerSecond;
108     }
109     else
110     {
111         ticks_per_second = 25.0f;
112     }
113
114     // retrieve the directory path of the filepath
115     directory = path.substr(0, path.find_last_of('/'));
116
117     cout << "scene->HasAnimations() 1: " << scene->HasAnimations() << endl;
118     cout << "scene->mNumMeshes 1: " << scene->mNumMeshes << endl;
119     cout << "scene->mAnimations[0]->mNumChannels 1: " << scene->mAnimations[0]->mNumChannels << endl;
120     cout << "scene->mAnimations[0]->mDuration 1: " << scene->mAnimations[0]->mDuration << endl;
121     cout << "scene->mAnimations[0]->mTicksPerSecond 1: " << scene->mAnimations[0]->mTicksPerSecond <<
122     endl << endl;
123
124     cout << "        name nodes : " << endl;
125     showNodeName(scene->mRootNode);
126     cout << endl;
127
128     cout << "        name bones : " << endl;
129     //processNode(scene->mRootNode, scene);
130     // process ASSIMP's root node recursively
131     processNode(scene->mRootNode, scene);
132
133     cout << "        name nodes animation : " << endl;
134     for (uint i = 0; i < scene->mAnimations[0]->mNumChannels; i++)
135     {

```

```

132         cout << scene->mAnimations[0]->mChannels[i]->mNodeName.C_Str() << endl;
133     }
134     cout << endl;
135 }
136
137 void showNodeName(aiNode* node)
138 {
139     cout << node->mName.data << endl;
140     for (uint i = 0; i < node->mNumChildren; i++)
141     {
142         showNodeName(node->mChildren[i]);
143     }
144 }
145
146 // processes a node in a recursive fashion. Processes each individual mesh located at the node and
147 // repeats this process on its children nodes (if any).
148 void processNode(aiNode *node, const aiScene *scene)
149 {
150     // process each mesh located at the current node
151     for (unsigned int i = 0; i < node->mNumMeshes; i++)
152     {
153         // the node object only contains indices to index the actual objects in the scene.
154         // the scene contains all the data, node is just to keep stuff organized (like relations
155         // between nodes).
156         aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
157         meshes.push_back(processMesh(mesh, scene));
158     }
159     // after we've processed all of the meshes (if any) we then recursively process each of the
160     // children nodes
161     for (unsigned int i = 0; i < node->mNumChildren; i++)
162     {
163         processNode(node->mChildren[i], scene);
164     }
165 }
166
167 MeshAnim processMesh(aiMesh *mesh, const aiScene *scene)
168 {
169     std::cout << "bones: " << mesh->mNumBones << " vertices: " << mesh->mNumVertices << std::endl;
170     // data to fill
171     vector<Vertex> vertices;
172     vector<unsigned int> indices;
173     vector<Texture> textures;
174     vector<VertexBoneData> bones_id_weights_for_each_vertex;
175
176     //Tal vez haya que hacer resize de los vectores
177     vertices.reserve(mesh->mNumVertices);
178     indices.reserve(mesh->mNumVertices);
179     bones_id_weights_for_each_vertex.resize(mesh->mNumVertices);
180
181     // Walk through each of the mesh's vertices
182     for (unsigned int i = 0; i < mesh->mNumVertices; i++)
183     {
184         Vertex vertex;
185         glm::vec3 vector; // we declare a placeholder vector since assimp uses its own vector class
186         // that doesn't directly convert to glm's vec3 class so we transfer the data to this placeholder
187         glm::vec3 first;
188         // positions
189         vector.x = mesh->mVertices[i].x;
190         vector.y = mesh->mVertices[i].y;
191         vector.z = mesh->mVertices[i].z;
192         vertex.Position = vector;
193         // normals
194         vector.x = mesh->mNormals[i].x;
195         vector.y = mesh->mNormals[i].y;
196         vector.z = mesh->mNormals[i].z;
197         vertex.Normal = vector;
198         // texture coordinates
199         if (mesh->mTextureCoords[0]) // does the mesh contain texture coordinates?
200         {
201             glm::vec2 vec;
202             // a vertex can contain up to 8 different texture coordinates. We thus make the
203             // assumption that we won't
204             // use models where a vertex can have multiple texture coordinates so we always take the
205             // first set (0).
206             vec.x = mesh->mTextureCoords[0][i].x;
207             vec.y = mesh->mTextureCoords[0][i].y;
208             vertex.TexCoords = vec;
209         }
210         else
211         {
212             vertex.TexCoords = glm::vec2(0.0f, 0.0f);
213             // tangent
214             vector.x = mesh->mTangents[i].x;
215             vector.y = mesh->mTangents[i].y;
216             vector.z = mesh->mTangents[i].z;

```



```

288         // return a mesh object created from the extracted mesh data
289         return MeshAnim(vertices, indices, textures, bones_id_weights_for_each_vertex);
290     }
291
292     // checks all material textures of a given type and loads the textures if they're not loaded yet.
293     // the required info is returned as a Texture struct.
294     vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type, string typeName)
295     {
296         vector<Texture> textures;
297         for(unsigned int i = 0; i < mat->GetTextureCount(type); i++)
298         {
299             aiString str;
300             mat->GetTexture(type, i, &str);
301             // check if texture was loaded before and if so, continue to next iteration: skip loading a
302             new texture
303             bool skip = false;
304             for(unsigned int j = 0; j < textures_loaded.size(); j++)
305             {
306                 if(std::strcmp(textures_loaded[j].path.data(), str.C_Str()) == 0)
307                 {
308                     textures.push_back(textures_loaded[j]);
309                     skip = true; // a texture with the same filepath has already been loaded, continue
310                     to next one. (optimization)
311                     break;
312                 }
313             }
314             if(!skip)
315             {
316                 // if texture hasn't been loaded already, load it
317                 Texture texture;
318                 texture.id = TextureFromFile(str.C_Str(), this->directory);
319                 texture.type = typeName;
320                 texture.path = str.C_Str();
321                 textures.push_back(texture);
322                 textures_loaded.push_back(texture); // store it as texture loaded for entire model, to
323                 ensure we won't unnecesary load duplicate textures.
324             }
325         }
326         return textures;
327     }
328
329     uint findPosition(float p_animation_time, const aiNodeAnim* p_node_anim)
330     {
331         // =====
332         for (uint i = 0; i < p_node_anim->mNumPositionKeys - 1; i++) // =====
333         {
334             if (p_animation_time < (float)p_node_anim->mPositionKeys[i + 1].mTime) // =====
335             {
336                 return i; // =====
337             }
338         }
339         assert(0);
340         return 0;
341     }
342
343     uint findRotation(float p_animation_time, const aiNodeAnim* p_node_anim)
344     {
345         // =====
346         for (uint i = 0; i < p_node_anim->mNumRotationKeys - 1; i++) // =====
347         {
348             if (p_animation_time < (float)p_node_anim->mRotationKeys[i + 1].mTime) // =====
349             {
350                 return i; // =====
351             }
352         }
353         assert(0);
354         return 0;
355     }
356
357     uint findScaling(float p_animation_time, const aiNodeAnim* p_node_anim)
358     {
359         // =====
360         for (uint i = 0; i < p_node_anim->mNumScalingKeys - 1; i++) // =====
361         {
362             if (p_animation_time < (float)p_node_anim->mScalingKeys[i + 1].mTime) // =====
363             {
364                 return i; // =====
365             }
366         }
367         assert(0);
368         return 0;
369     }

```

```

369
370     aiVector3D calcInterpolatedPosition(float p_animation_time, const aiNodeAnim* p_node_anim)
371     {
372         if (p_node_anim->mNumPositionKeys == 1) // Keys ??? ?????? ?????
373         {
374             return p_node_anim->mPositionKeys[0].mValue;
375         }
376
377         uint position_index = findPosition(p_animation_time, p_node_anim); // ?????? ?????? ??????
378         uint next_position_index = position_index + 1; // ?????? ?????? ?????? ??????
379         assert(next_position_index < p_node_anim->mNumPositionKeys);
380         // ?????? ?????? ??????
381         float delta_time = (float)(p_node_anim->mPositionKeys[next_position_index].mTime -
382 p_node_anim->mPositionKeys[position_index].mTime);
383         // ?????? = (????? ?????? ?????? ?? ?????? ?????? ?????? ??????) / ?? ??? ?????? ??????
384         float factor = (p_animation_time - (float)p_node_anim->mPositionKeys[position_index].mTime) /
385 delta_time;
386         assert(factor >= 0.0f && factor <= 1.0f);
387         aiVector3D start = p_node_anim->mPositionKeys[position_index].mValue;
388         aiVector3D end = p_node_anim->mPositionKeys[next_position_index].mValue;
389         aiVector3D delta = end - start;
390
391         return start + factor * delta;
392     }
393
394     aiQuaternion calcInterpolatedRotation(float p_animation_time, const aiNodeAnim* p_node_anim)
395     {
396         if (p_node_anim->mNumRotationKeys == 1) // Keys ??? ?????? ?????
397         {
398             return p_node_anim->mRotationKeys[0].mValue;
399         }
400
401         uint rotation_index = findRotation(p_animation_time, p_node_anim); // ?????? ?????? ??????
402         uint next_rotation_index = rotation_index + 1; // ?????? ?????? ?????? ??????
403         assert(next_rotation_index < p_node_anim->mNumRotationKeys);
404         // ?????? ?????? ??????
405         float delta_time = (float)(p_node_anim->mRotationKeys[next_rotation_index].mTime -
406 p_node_anim->mRotationKeys[rotation_index].mTime);
407         // ?????? = (????? ?????? ?????? ?? ?????? ?????? ?????? ??????) / ?? ??? ?????? ??????
408         float factor = (p_animation_time - (float)p_node_anim->mRotationKeys[rotation_index].mTime) /
409 delta_time;
410
411         //cout << "p_node_anim->mRotationKeys[rotation_index].mTime: " <<
412 p_node_anim->mRotationKeys[rotation_index].mTime << endl;
413         //cout << "p_node_anim->mRotationKeys[next_rotaion_index].mTime: " <<
414 p_node_anim->mRotationKeys[next_rotation_index].mTime << endl;
415         //cout << "delta_time: " << delta_time << endl;
416         //cout << "animation_time: " << p_animation_time << endl;
417         //cout << "animation_time - mRotationKeys[rotation_index].mTime: " << (p_animation_time -
418 (float)p_node_anim->mRotationKeys[rotation_index].mTime) << endl;
419         //cout << "factor: " << factor << endl << endl << endl;
420
421         assert(factor >= 0.0f && factor <= 1.0f);
422         aiQuaternion start_quat = p_node_anim->mRotationKeys[rotation_index].mValue;
423         aiQuaternion end_quat = p_node_anim->mRotationKeys[next_rotation_index].mValue;
424
425         return nlerp(start_quat, end_quat, factor);
426     }
427
428     aiVector3D calcInterpolatedScaling(float p_animation_time, const aiNodeAnim* p_node_anim)
429     {
430         if (p_node_anim->mNumScalingKeys == 1) // Keys ??? ?????? ?????
431         {
432             return p_node_anim->mScalingKeys[0].mValue;
433         }
434
435         uint scaling_index = findScaling(p_animation_time, p_node_anim); // ?????? ?????? ?????? ??????
436         uint next_scaling_index = scaling_index + 1; // ?????? ?????? ?????? ??????
437         assert(next_scaling_index < p_node_anim->mNumScalingKeys);
438         // ?????? ?????? ??????
439         float delta_time = (float)(p_node_anim->mScalingKeys[next_scaling_index].mTime -
440 p_node_anim->mScalingKeys[scaling_index].mTime);
441         // ?????? = (????? ?????? ?????? ?? ?????? ?????? ?????? ??????) / ?? ??? ?????? ??????
442         float factor = (p_animation_time - (float)p_node_anim->mScalingKeys[scaling_index].mTime) /
443 delta_time;
444         //cout << "p_animation_time: " << p_animation_time << " " << "mTime: " <<
445 (float)p_node_anim->mScalingKeys[scaling_index].mTime << endl << endl << endl;
446         assert(factor >= 0.0f && factor <= 1.0f);
447         aiVector3D start = p_node_anim->mScalingKeys[scaling_index].mValue;
448         aiVector3D end = p_node_anim->mScalingKeys[next_scaling_index].mValue;
449         aiVector3D delta = end - start;
450
451         return start + factor * delta;
452     }

```



```

444     const aiNodeAnim * findNodeAnim(const aiAnimation * p_animation, const string p_node_name)
445     {
446         // channel in animation contains aiNodeAnim (aiNodeAnim its transformation for bones)
447         // numChannels == numBones
448         for (uint i = 0; i < p_animation->mNumChannels; i++)
449         {
450             const aiNodeAnim* node_anim = p_animation->mChannels[i]; // ***** node
451             if (string(node_anim->mNodeName.data) == p_node_name)
452             {
453                 return node_anim; // ***** (* ***** node)
454             }
455         }
456
457         return nullptr;
458     }
459
460     // start from RootNode
461     void readNodeHierarchy(float p_animation_time, const aiNode* p_node, const aiMatrix4x4
462     parent_transform)
463     {
464         string node_name(p_node->mName.data);
465
466         //***** node, ** ***** bone, ***** (aiNodeAnim).
467         const aiAnimation* animation = scene->mAnimations[0];
468         aiMatrix4x4 node_transform = p_node->mTransformation; // AQU
469         AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
470
471         const aiNodeAnim* node_anim = findNodeAnim(animation, node_name); // *****
472
473         if (node_anim)
474         {
475             //scaling
476             //aiVector3D scaling_vector = node_anim->mScalingKeys[2].mValue;
477             aiVector3D scaling_vector = calcInterpolatedScaling(p_animation_time, node_anim);
478             aiMatrix4x4 scaling_matr;
479             aiMatrix4x4::Scaling(scaling_vector, scaling_matr);
480
481             //rotation
482             //aiQuaternion rotate_quat = node_anim->mRotationKeys[2].mValue;
483             aiQuaternion rotate_quat = calcInterpolatedRotation(p_animation_time, node_anim);
484             aiMatrix4x4 rotate_matr = aiMatrix4x4(rotate_quat.GetMatrix());
485
486             //translation
487             //aiVector3D translate_vector = node_anim->mPositionKeys[2].mValue;
488             aiVector3D translate_vector = calcInterpolatedPosition(p_animation_time, node_anim);
489             aiMatrix4x4 translate_matr;
490             aiMatrix4x4::Translation(translate_vector, translate_matr);
491
492             //if (p_node->mName == scene->mRootNode->mName) {
493             //    node_transform = translate_matr * (rotate_matr * aiMatrix4x4 (aiQuaternion(-90.0f, 0.0f,
494             0.0f).GetMatrix())) * scaling_matr;
495             //}
496
497             node_transform = translate_matr * rotate_matr * scaling_matr;
498         }
499
500         aiMatrix4x4 global_transform = parent_transform * node_transform;
501
502         // **** * node ** ***** bone, ** * node ***** * ***** bone !!!
503         if (m_bone_mapping.find(node_name) != m_bone_mapping.end()) // true if node_name exist in
504         bone_mapping
505         {
506             uint bone_index = m_bone_mapping[node_name];
507             m_bone_matrices[bone_index].final_world_transform = m_global_inverse_transform *
508             global_transform * m_bone_matrices[bone_index].offset_matrix;
509         }
510
511         for (uint i = 0; i < p_node->mNumChildren; i++)
512         {
513             readNodeHierarchy(p_animation_time, p_node->mChildren[i], global_transform);
514         }
515
516     }
517
518     void boneTransform(double time_in_sec, vector<aiMatrix4x4>& transforms)
519     {
520         aiMatrix4x4 identity_matrix; // = mat4(1.0f);
521         double time_in_ticks = time_in_sec * ticks_per_second;
522         float animation_time = fmod(time_in_ticks, scene->mAnimations[0]->mDuration); //*****
523         (*****
524         // animation_time - **** ***** * **** ***** (* ***** )
525         ***** )

```

```

522     readNodeHierarchy(animation_time, scene->mRootNode, identity_matrix);
523
524     transforms.resize(m_num_bones);
525
526     for (uint i = 0; i < m_num_bones; i++)
527     {
528         transforms[i] = m_bone_matrices[i].final_world_transform;
529     }
530 }
531
532 glm::mat4 aiToGlm(aiMatrix4x4 ai_matr)
533 {
534     glm::mat4 result;
535     result[0].x = ai_matr.a1; result[0].y = ai_matr.b1; result[0].z = ai_matr.c1; result[0].w =
ai_matr.d1;
536     result[1].x = ai_matr.a2; result[1].y = ai_matr.b2; result[1].z = ai_matr.c2; result[1].w =
ai_matr.d2;
537     result[2].x = ai_matr.a3; result[2].y = ai_matr.b3; result[2].z = ai_matr.c3; result[2].w =
ai_matr.d3;
538     result[3].x = ai_matr.a4; result[3].y = ai_matr.b4; result[3].z = ai_matr.c4; result[3].w =
ai_matr.d4;
539
540     //cout << " " << result[0].x << "          " << result[0].y << "          " << result[0].z << "          " <<
result[0].w << endl;
541     //cout << " " << result[1].x << "          " << result[1].y << "          " << result[1].z << "          " <<
result[1].w << endl;
542     //cout << " " << result[2].x << "          " << result[2].y << "          " << result[2].z << "          " <<
result[2].w << endl;
543     //cout << " " << result[3].x << "          " << result[3].y << "          " << result[3].z << "          " <<
result[3].w << endl;
544     //cout << endl;
545
546     //cout << " " << ai_matr.a1 << "          " << ai_matr.b1 << "          " << ai_matr.c1 << "          " <<
ai_matr.d1 << endl;
547     //cout << " " << ai_matr.a2 << "          " << ai_matr.b2 << "          " << ai_matr.c2 << "          " <<
ai_matr.d2 << endl;
548     //cout << " " << ai_matr.a3 << "          " << ai_matr.b3 << "          " << ai_matr.c3 << "          " <<
ai_matr.d3 << endl;
549     //cout << " " << ai_matr.a4 << "          " << ai_matr.b4 << "          " << ai_matr.c4 << "          " <<
ai_matr.d4 << endl;
550     //cout << endl;
551
552     return result;
553 }
554
555
556 aiQuaternion nlerp(aiQuaternion a, aiQuaternion b, float blend)
557 {
558     //cout << a.w + a.x + a.y + a.z << endl;
559     a.Normalize();
560     b.Normalize();
561
562     aiQuaternion result;
563     float dot_product = a.x * b.x + a.y * b.y + a.z * b.z + a.w * b.w;
564     float one_minus_blend = 1.0f - blend;
565
566     if (dot_product < 0.0f)
567     {
568         result.x = a.x * one_minus_blend + blend * -b.x;
569         result.y = a.y * one_minus_blend + blend * -b.y;
570         result.z = a.z * one_minus_blend + blend * -b.z;
571         result.w = a.w * one_minus_blend + blend * -b.w;
572     }
573     else
574     {
575         result.x = a.x * one_minus_blend + blend * b.x;
576         result.y = a.y * one_minus_blend + blend * b.y;
577         result.z = a.z * one_minus_blend + blend * b.z;
578         result.w = a.w * one_minus_blend + blend * b.w;
579     }
580
581     return result.Normalize();
582 }
583 };
584 #endif

```

4.7 Shader.h

```

1 #ifndef SHADER_H
2 #define SHADER_H
3
4 #include <string>
5 #include <fstream>

```

```

6 #include <sstream>
7 #include <iostream>
8
9 #include <GL/glew.h>
10
11 class Shader
12 {
13 public:
14     GLuint Program;
15     GLuint uniformColor;
16     // Constructor generates the shader on the fly
17     Shader(const GLchar *vertexPath, const GLchar *fragmentPath)
18     {
19         // 1. Retrieve the vertex/fragment source code from filePath
20         std::string vertexCode;
21         std::string fragmentCode;
22         std::ifstream vShaderFile;
23         std::ifstream fShaderFile;
24         // ensures ifstream objects can throw exceptions:
25         vShaderFile.exceptions(std::ifstream::badbit);
26         fShaderFile.exceptions(std::ifstream::badbit);
27         try
28         {
29             // Open files
30             vShaderFile.open(vertexPath);
31             fShaderFile.open(fragmentPath);
32             std::stringstream vShaderStream, fShaderStream;
33             // Read file's buffer contents into streams
34             vShaderStream << vShaderFile.rdbuf();
35             fShaderStream << fShaderFile.rdbuf();
36             // close file handlers
37             vShaderFile.close();
38             fShaderFile.close();
39             // Convert stream into string
40             vertexCode = vShaderStream.str();
41             fragmentCode = fShaderStream.str();
42         }
43         catch (std::ifstream::failure e)
44         {
45             std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
46         }
47         const GLchar *vShaderCode = vertexCode.c_str();
48         const GLchar *fShaderCode = fragmentCode.c_str();
49         // 2. Compile shaders
50         GLuint vertex, fragment;
51         GLint success;
52         GLchar infoLog[512];
53         // Vertex Shader
54         vertex = glCreateShader(GL_VERTEX_SHADER);
55         glShaderSource(vertex, 1, &vShaderCode, NULL);
56         glCompileShader(vertex);
57         // Print compile errors if any
58         glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
59         if (!success)
60         {
61             glGetShaderInfoLog(vertex, 512, NULL, infoLog);
62             std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
63         }
64         // Fragment Shader
65         fragment = glCreateShader(GL_FRAGMENT_SHADER);
66         glShaderSource(fragment, 1, &fShaderCode, NULL);
67         glCompileShader(fragment);
68         // Print compile errors if any
69         glGetShaderiv(fragment, GL_COMPILE_STATUS, &success);
70         if (!success)
71         {
72             glGetShaderInfoLog(fragment, 512, NULL, infoLog);
73             std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
74         }
75         // Shader Program
76         this->Program = glCreateProgram();
77         glAttachShader(this->Program, vertex);
78         glAttachShader(this->Program, fragment);
79         glLinkProgram(this->Program);
80         // Print linking errors if any
81         glGetProgramiv(this->Program, GL_LINK_STATUS, &success);
82         if (!success)
83         {
84             glGetProgramInfoLog(this->Program, 512, NULL, infoLog);
85             std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
86         }
87         //le damos la localidad de color
88         uniformColor = glGetUniformLocation(this->Program, "color");
89         // Delete the shaders as they're linked into our program now and no longer necessary
90         glDeleteShader(vertex);
91         glDeleteShader(fragment);
92     }

```

```

93     }
94     // Uses the current shader
95     void Use()
96     {
97         glUseProgram(this->Program);
98     }
99
100     GLuint getUniformLocation()
101     {
102         return uniformColor;
103     }
104 };
105
106 #endif

```

4.8 stb_image.h

```

1  /* stb_image - v2.14 - public domain image loader - http://nothings.org/stb_image.h
2  no warranty implied; use at your own risk
3
4  Do this:
5  #define STB_IMAGE_IMPLEMENTATION
6  before you include this file in *one* C or C++ file to create the implementation.
7
8  // i.e. it should look like this:
9  #include ...
10 #include ...
11 #include ...
12 #define STB_IMAGE_IMPLEMENTATION
13 #include "stb_image.h"
14
15 You can #define STBI_ASSERT(x) before the #include to avoid using assert.h.
16 And #define STBI_MALLOC, STBI_REALLOC, and STBI_FREE to avoid using malloc, realloc, free
17
18
19 QUICK NOTES:
20 Primarily of interest to game developers and other people who can
21 avoid problematic images and only need the trivial interface
22
23 JPEG baseline & progressive (12 bpc/arithmetic not supported, same as stock IJG lib)
24 PNG 1/2/4/8-bit-per-channel (16 bpc not supported)
25
26 TGA (not sure what subset, if a subset)
27 BMP non-lbpp, non-RLE
28 PSD (composited view only, no extra channels, 8/16 bit-per-channel)
29
30 GIF (*comp always reports as 4-channel)
31 HDR (radiance rgbE format)
32 PIC (Softimage PIC)
33 PNM (PPM and PGM binary only)
34
35 Animated GIF still needs a proper API, but here's one way to do it:
36 http://gist.github.com/urraka/685d9a6340b26b830d49
37
38 - decode from memory or through FILE (define STBI_NO_STDIO to remove code)
39 - decode from arbitrary I/O callbacks
40 - SIMD acceleration on x86/x64 (SSE2) and ARM (NEON)
41
42 Full documentation under "DOCUMENTATION" below.
43
44
45 Revision 2.00 release notes:
46
47 - Progressive JPEG is now supported.
48
49 - PPM and PGM binary formats are now supported, thanks to Ken Miller.
50
51 - x86 platforms now make use of SSE2 SIMD instructions for
52 JPEG decoding, and ARM platforms can use NEON SIMD if requested.
53 This work was done by Fabian "ryg" Giesen. SSE2 is used by
54 default, but NEON must be enabled explicitly; see docs.
55
56 With other JPEG optimizations included in this version, we see
57 2x speedup on a JPEG on an x86 machine, and a 1.5x speedup
58 on a JPEG on an ARM machine, relative to previous versions of this
59 library. The same results will not obtain for all JPGs and for all
60 x86/ARM machines. (Note that progressive JPEGs are significantly
61 slower to decode than regular JPEGs.) This doesn't mean that this
62 is the fastest JPEG decoder in the land; rather, it brings it
63 closer to parity with standard libraries. If you want the fastest
64 decode, look elsewhere. (See "Philosophy" section of docs below.)
65
66 See final bullet items below for more info on SIMD.

```

```

67
68 - Added STBI_MALLOC, STBI_REALLOC, and STBI_FREE macros for replacing
69 the memory allocator. Unlike other STBI libraries, these macros don't
70 support a context parameter, so if you need to pass a context in to
71 the allocator, you'll have to store it in a global or a thread-local
72 variable.
73
74 - Split existing STBI_NO_HDR flag into two flags, STBI_NO_HDR and
75 STBI_NO_LINEAR.
76 STBI_NO_HDR:    suppress implementation of .hdr reader format
77 STBI_NO_LINEAR: suppress high-dynamic-range light-linear float API
78
79 - You can suppress implementation of any of the decoders to reduce
80 your code footprint by #defining one or more of the following
81 symbols before creating the implementation.
82
83 STBI_NO_JPEG
84 STBI_NO_PNG
85 STBI_NO_BMP
86 STBI_NO_PSD
87 STBI_NO_TGA
88 STBI_NO_GIF
89 STBI_NO_HDR
90 STBI_NO_PIC
91 STBI_NO_PNM    (.ppm and .pgm)
92
93 - You can request *only* certain decoders and suppress all other ones
94 (this will be more forward-compatible, as addition of new decoders
95 doesn't require you to disable them explicitly):
96
97 STBI_ONLY_JPEG
98 STBI_ONLY_PNG
99 STBI_ONLY_BMP
100 STBI_ONLY_PSD
101 STBI_ONLY_TGA
102 STBI_ONLY_GIF
103 STBI_ONLY_HDR
104 STBI_ONLY_PIC
105 STBI_ONLY_PNM    (.ppm and .pgm)
106
107 Note that you can define multiples of these, and you will get all
108 of them ("only x" and "only y" is interpreted to mean "only x&y").
109
110 - If you use STBI_NO_PNG (or _ONLY_ without PNG), and you still
111 want the zlib decoder to be available, #define STBI_SUPPORT_ZLIB
112
113 - Compilation of all SIMD code can be suppressed with
114 #define STBI_NO_SIMD
115 It should not be necessary to disable SIMD unless you have issues
116 compiling (e.g. using an x86 compiler which doesn't support SSE
117 intrinsics or that doesn't support the method used to detect
118 SSE2 support at run-time), and even those can be reported as
119 bugs so I can refine the built-in compile-time checking to be
120 smarter.
121
122 - The old STBI_SIMD system which allowed installing a user-defined
123 IDCT etc. has been removed. If you need this, don't upgrade. My
124 assumption is that almost nobody was doing this, and those who
125 were will find the built-in SIMD more satisfactory anyway.
126
127 - RGB values computed for JPEG images are slightly different from
128 previous versions of stb_image. (This is due to using less
129 integer precision in SIMD.) The C code has been adjusted so
130 that the same RGB values will be computed regardless of whether
131 SIMD support is available, so your app should always produce
132 consistent results. But these results are slightly different from
133 previous versions. (Specifically, about 3% of available YCbCr values
134 will compute different RGB results from pre-1.49 versions by +-1;
135 most of the deviating values are one smaller in the G channel.)
136
137 - If you must produce consistent results with previous versions of
138 stb_image, #define STBI_JPEG_OLD and you will get the same results
139 you used to; however, you will not get the SIMD speedups for
140 the YCbCr-to-RGB conversion step (although you should still see
141 significant JPEG speedup from the other changes).
142
143 Please note that STBI_JPEG_OLD is a temporary feature; it will be
144 removed in future versions of the library. It is only intended for
145 near-term back-compatibility use.
146
147
148 Latest revision history:
149 2.13  (2016-12-04) experimental 16-bit API, only for PNG so far; fixes
150 2.12  (2016-04-02) fix typo in 2.11 PSD fix that caused crashes
151 2.11  (2016-04-02) 16-bit PNGs; enable SSE2 in non-gcc x64
152 RGB-format JPEG; remove white matting in PSD;
153 allocate large structures on the stack;

```

```

154 correct channel count for PNG & BMP
155 2.10 (2016-01-22) avoid warning introduced in 2.09
156 2.09 (2016-01-16) 16-bit TGA; comments in PNM files; STBI_REALLOC_SIZED
157 2.08 (2015-09-13) fix to 2.07 cleanup, reading RGB PSD as RGBA
158 2.07 (2015-09-13) partial animated GIF support
159 limited 16-bit PSD support
160 minor bugs, code cleanup, and compiler warnings
161
162 See end of file for full revision history.
163
164
165 ===== Contributors =====
166
167 Image formats                                Extensions, features
168 Sean Barrett (jpeg, png, bmp)                Jetro Lauha (stbi_info)
169 Nicolas Schulz (hdr, psd)                     Martin "SpartanJ" Golini (stbi_info)
170 Jonathan Dummer (tga)                        James "moose2000" Brown (iPhone PNG)
171 Jean-Marc Lienher (gif)                      Ben "Disch" Wenger (io callbacks)
172 Tom Seddon (pic)                             Omar Cornut (1/2/4-bit PNG)
173 Thatcher Ulrich (psd)                       Nicolas Guillemot (vertical flip)
174 Ken Miller (pgm, ppm)                       Richard Mitton (16-bit PSD)
175 github:urraka (animated gif)                 Junggon Kim (PNM comments)
176 Daniel Gibson (16-bit TGA)
177 socks-the-fox (16-bit TGA)
178 Optimizations & bugfixes
179 Fabian "ryg" Giesen
180 Arseny Kapoulkine
181
182 Bug & warning fixes
183 Marc LeBlanc                                David Woo                Guillaume George          Martins Mozeiko
184 Christopher Lloyd                            Martin Golini            Jerry Jansson             Joseph Thomson
185 Dave Moore                                  Roy Eltham              Hayaki Saito              Phil Jordan
186 Won Chun                                    Luke Graham             Johan Duparc              Nathan Reed
187 the Horde3D community                       Thomas Ruf              Ronny Chevalier           Nick Verigakis
188 Janez Zemva                                John Bartholomew        Michal Cichon             github:svdijk
189 Jonathan Blow                              Ken Hamada              Tero Hanninen            Baldur Karlsson
190 Laurent Gomila                             Cort Stratton           Sergio Gonzalez           github:romigrou
191 Aruelien Pocheville                       Thibault Reuille       Cass Everitt              Matthew Gregan
192 Ryamond Barbiero                          Paul Du Bois            Engin Manap               github:snagar
193 Michaelangel007@github                    Oriol Ferrer Mesia     Dale Weiler               github:Zelex
194 Philipp Wiesemann                         Josh Tobin              github:rlyeh               github:grim210@github
195 Blazej Dariusz Roszkowski                  github:sammyhw
196
197
198 LICENSE
199
200 This software is dual-licensed to the public domain and under the following
201 license: you are granted a perpetual, irrevocable license to copy, modify,
202 publish, and distribute this file as you see fit.
203
204 */
205
206 #ifndef STBI_INCLUDE_STB_IMAGE_H
207 #define STBI_INCLUDE_STB_IMAGE_H
208
209 // DOCUMENTATION
210 //
211 // Limitations:
212 //   - no 16-bit-per-channel PNG
213 //   - no 12-bit-per-channel JPEG
214 //   - no JPEGs with arithmetic coding
215 //   - no 1-bit BMP
216 //   - GIF always returns *comp=4
217 //
218 // Basic usage (see HDR discussion below for HDR usage):
219 //   int x,y,n;
220 //   unsigned char *data = stbi_load(filename, &x, &y, &n, 0);
221 //   // ... process data if not NULL ...
222 //   // ... x = width, y = height, n = # 8-bit components per pixel ...
223 //   // ... replace '0' with '1'..'4' to force that many components per pixel
224 //   // ... but 'n' will always be the number that it would have been if you said 0
225 //   stbi_image_free(data)
226 //
227 // Standard parameters:
228 //   int *x                -- outputs image width in pixels
229 //   int *y                -- outputs image height in pixels
230 //   int *channels_in_file  -- outputs # of image components in image file
231 //   int desired_channels   -- if non-zero, # of image components requested in result
232 //
233 // The return value from an image loader is an 'unsigned char *' which points
234 // to the pixel data, or NULL on an allocation failure or if the image is
235 // corrupt or invalid. The pixel data consists of *y scanlines of *x pixels,
236 // with each pixel consisting of N interleaved 8-bit components; the first
237 // pixel pointed to is top-left-most in the image. There is no padding between
238 // image scanlines or between pixels, regardless of format. The number of
239 // components N is 'req_comp' if req_comp is non-zero, or *comp otherwise.
240 // If req_comp is non-zero, *comp has the number of components that _would_

```

```

241 // have been output otherwise. E.g. if you set req_comp to 4, you will always
242 // get RGBA output, but you can check *comp to see if it's trivially opaque
243 // because e.g. there were only 3 channels in the source image.
244 //
245 // An output image with N components has the following components interleaved
246 // in this order in each pixel:
247 //
248 //      N=#comp      components
249 //      1            grey
250 //      2            grey, alpha
251 //      3            red, green, blue
252 //      4            red, green, blue, alpha
253 //
254 // If image loading fails for any reason, the return value will be NULL,
255 // and *x, *y, *comp will be unchanged. The function stbi_failure_reason()
256 // can be queried for an extremely brief, end-user unfriendly explanation
257 // of why the load failed. Define STBI_NO_FAILURE_STRINGS to avoid
258 // compiling these strings at all, and STBI_FAILURE_USERMSG to get slightly
259 // more user-friendly ones.
260 //
261 // Paletted PNG, BMP, GIF, and PIC images are automatically depalettized.
262 //
263 // =====
264 //
265 // Philosophy
266 //
267 // stb libraries are designed with the following priorities:
268 //
269 //      1. easy to use
270 //      2. easy to maintain
271 //      3. good performance
272 //
273 // Sometimes I let "good performance" creep up in priority over "easy to maintain",
274 // and for best performance I may provide less-easy-to-use APIs that give higher
275 // performance, in addition to the easy to use ones. Nevertheless, it's important
276 // to keep in mind that from the standpoint of you, a client of this library,
277 // all you care about is #1 and #3, and stb libraries do not emphasize #3 above all.
278 //
279 // Some secondary priorities arise directly from the first two, some of which
280 // make more explicit reasons why performance can't be emphasized.
281 //
282 //      - Portable ("ease of use")
283 //      - Small footprint ("easy to maintain")
284 //      - No dependencies ("ease of use")
285 //
286 // =====
287 //
288 // I/O callbacks
289 //
290 // I/O callbacks allow you to read from arbitrary sources, like packaged
291 // files or some other source. Data read from callbacks are processed
292 // through a small internal buffer (currently 128 bytes) to try to reduce
293 // overhead.
294 //
295 // The three functions you must define are "read" (reads some bytes of data),
296 // "skip" (skips some bytes of data), "eof" (reports if the stream is at the end).
297 //
298 // =====
299 //
300 // SIMD support
301 //
302 // The JPEG decoder will try to automatically use SIMD kernels on x86 when
303 // supported by the compiler. For ARM Neon support, you must explicitly
304 // request it.
305 //
306 // (The old do-it-yourself SIMD API is no longer supported in the current
307 // code.)
308 //
309 // On x86, SSE2 will automatically be used when available based on a run-time
310 // test; if not, the generic C versions are used as a fall-back. On ARM targets,
311 // the typical path is to have separate builds for NEON and non-NEON devices
312 // (at least this is true for iOS and Android). Therefore, the NEON support is
313 // toggled by a build flag: define STBI_NEON to get NEON loops.
314 //
315 // The output of the JPEG decoder is slightly different from versions where
316 // SIMD support was introduced (that is, for versions before 1.49). The
317 // difference is only +-1 in the 8-bit RGB channels, and only on a small
318 // fraction of pixels. You can force the pre-1.49 behavior by defining
319 // STBI_JPEG_OLD, but this will disable some of the SIMD decoding path
320 // and hence cost some performance.
321 //
322 // If for some reason you do not want to use any of SIMD code, or if
323 // you have issues compiling it, you can disable it entirely by
324 // defining STBI_NO_SIMD.
325 //
326 // =====
327 //

```

```

328 // HDR image support    (disable by defining STBI_NO_HDR)
329 //
330 // stbi_image now supports loading HDR images in general, and currently
331 // the Radiance .HDR file format, although the support is provided
332 // generically. You can still load any file through the existing interface;
333 // if you attempt to load an HDR file, it will be automatically remapped to
334 // LDR, assuming gamma 2.2 and an arbitrary scale factor defaulting to 1;
335 // both of these constants can be reconfigured through this interface:
336 //
337 //     stbi_hdr_to_ldr_gamma(2.2f);
338 //     stbi_hdr_to_ldr_scale(1.0f);
339 //
340 // (note, do not use _inverse_ constants; stbi_image will invert them
341 // appropriately).
342 //
343 // Additionally, there is a new, parallel interface for loading files as
344 // (linear) floats to preserve the full dynamic range:
345 //
346 //     float *data = stbi_loadf(filename, &x, &y, &n, 0);
347 //
348 // If you load LDR images through this interface, those images will
349 // be promoted to floating point values, run through the inverse of
350 // constants corresponding to the above:
351 //
352 //     stbi_ldr_to_hdr_scale(1.0f);
353 //     stbi_ldr_to_hdr_gamma(2.2f);
354 //
355 // Finally, given a filename (or an open file or memory block--see header
356 // file for details) containing image data, you can query for the "most
357 // appropriate" interface to use (that is, whether the image is HDR or
358 // not), using:
359 //
360 //     stbi_is_hdr(char *filename);
361 //
362 // =====
363 //
364 // iPhone PNG support:
365 //
366 // By default we convert iphone-formatted PNGs back to RGB, even though
367 // they are internally encoded differently. You can disable this conversion
368 // by calling stbi_convert_iphone_png_to_rgb(0), in which case
369 // you will always just get the native iphone "format" through (which
370 // is BGR stored in RGB).
371 //
372 // Call stbi_set_unpremultiply_on_load(1) as well to force a divide per
373 // pixel to remove any premultiplied alpha *only* if the image file explicitly
374 // says there's premultiplied data (currently only happens in iPhone images,
375 // and only if iPhone convert-to-rgb processing is on).
376 //
377 //
378 //
379 #ifndef STBI_NO_STDIO
380 #include <stdio.h>
381 #endif // STBI_NO_STDIO
382
383 #define STBI_VERSION 1
384
385 enum
386 {
387     STBI_default = 0, // only used for req_comp
388
389     STBI_grey = 1,
390     STBI_grey_alpha = 2,
391     STBI_rgb = 3,
392     STBI_rgb_alpha = 4
393 };
394
395 typedef unsigned char stbi_uc;
396 typedef unsigned short stbi_us;
397
398 #ifdef __cplusplus
399 extern "C" {
400 #endif
401
402 #ifdef STB_IMAGE_STATIC
403 #define STBIDEF static
404 #else
405 #define STBIDEF extern
406 #endif
407
408 //
409 // PRIMARY API - works on images of any type
410 //
411 //
412 //
413 //
414 // load image by filename, open file, or memory buffer
415 //

```



```

416
417     typedef struct
418     {
419         int(*read)  (void *user, char *data, int size);    // fill 'data' with 'size' bytes. return
number of bytes actually read
420         void(*skip) (void *user, int n);                  // skip the next 'n' bytes, or 'unget' the
last -n bytes if negative
421         int(*eof)   (void *user);                          // returns nonzero if we are at end of file/data
422     } stbi_io_callbacks;
423
424     //
425     // 8-bits-per-channel interface
426     //
427
428     STBIDEF stbi_uc *stbi_load(char          const *filename, int *x, int *y, int *channels_in_file,
int desired_channels);
429     STBIDEF stbi_uc *stbi_load_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int
*channels_in_file, int desired_channels);
430     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);
431
432 #ifndef STBI_NO_STDIO
433     STBIDEF stbi_uc *stbi_load_from_file(FILE *f, int *x, int *y, int *channels_in_file, int
desired_channels);
434     // for stbi_load_from_file, file pointer is left pointing immediately after image
435 #endif
436
437     //
438     // 16-bits-per-channel interface
439     //
440
441     STBIDEF stbi_us *stbi_load_16(char const *filename, int *x, int *y, int *channels_in_file, int
desired_channels);
442 #ifndef STBI_NO_STDIO
443     STBIDEF stbi_us *stbi_load_from_file_16(FILE *f, int *x, int *y, int *channels_in_file, int
desired_channels);
444 #endif
445     // @TODO the other variants
446
447     //
448     // float-per-channel interface
449     //
450
451 #ifndef STBI_NO_LINEAR
452     STBIDEF float *stbi_loadf(char const *filename, int *x, int *y, int *channels_in_file, int
desired_channels);
453     STBIDEF float *stbi_loadf_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int
*channels_in_file, int desired_channels);
454     STBIDEF float *stbi_loadf_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y,
int *channels_in_file, int desired_channels);
455 #endif
456 #ifndef STBI_NO_STDIO
457     STBIDEF float *stbi_loadf_from_file(FILE *f, int *x, int *y, int *channels_in_file, int
desired_channels);
458 #endif
459
460 #ifndef STBI_NO_HDR
461     STBIDEF void    stbi_hdr_to_ldr_gamma(float gamma);
462     STBIDEF void    stbi_ldr_to_hdr_gamma(float gamma);
463     STBIDEF void    stbi_hdr_to_ldr_scale(float scale);
464     STBIDEF void    stbi_ldr_to_hdr_scale(float scale);
465 #endif // STBI_NO_HDR
466
467 #ifndef STBI_NO_LINEAR
468     STBIDEF void    stbi_ldr_to_hdr_gamma(float gamma);
469     STBIDEF void    stbi_hdr_to_ldr_gamma(float gamma);
470     STBIDEF void    stbi_ldr_to_hdr_scale(float scale);
471     STBIDEF void    stbi_hdr_to_ldr_scale(float scale);
472 #endif // STBI_NO_LINEAR
473
474     // stbi_is_hdr is always defined, but always returns false if STBI_NO_HDR
475     STBIDEF int    stbi_is_hdr_from_callbacks(stbi_io_callbacks const *clbk, void *user);
476     STBIDEF int    stbi_is_hdr_from_memory(stbi_uc const *buffer, int len);
477 #ifndef STBI_NO_STDIO
478     STBIDEF int    stbi_is_hdr(char const *filename);
479     STBIDEF int    stbi_is_hdr_from_file(FILE *f);
480 #endif // STBI_NO_STDIO
481
482     // get a VERY brief reason for failure
483     // NOT THREADSAFE
484     STBIDEF const char *stbi_failure_reason(void);
485
486     // free the loaded image -- this is just free()
487     STBIDEF void    stbi_image_free(void *retval_from_stbi_load);
488
489     // get image dimensions & components without fully decoding
490     STBIDEF int    stbi_info_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp);
491     STBIDEF int    stbi_info_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y,
int *comp);
492

```

```

493 #ifndef STBI_NO_STDIO
494     STBIDEF int      stbi_info(char const *filename, int *x, int *y, int *comp);
495     STBIDEF int      stbi_info_from_file(FILE *f, int *x, int *y, int *comp);
496
497 #endif
498
499
500
501     // for image formats that explicitly notate that they have premultiplied alpha,
502     // we just return the colors as stored in the file. set this flag to force
503     // unpremultiplication. results are undefined if the unpremultiply overflow.
504     STBIDEF void stbi_set_unpremultiply_on_load(int flag_true_if_should_unpremultiply);
505
506     // indicate whether we should process iphone images back to canonical format,
507     // or just pass them through "as-is"
508     STBIDEF void stbi_convert_iphone_png_to_rgb(int flag_true_if_should_convert);
509
510     // flip the image vertically, so the first pixel in the output array is the bottom left
511     STBIDEF void stbi_set_flip_vertically_on_load(int flag_true_if_should_flip);
512
513     // ZLIB client - used by PNG, available for other purposes
514
515     STBIDEF char *stbi_zlib_decode_malloc_guesssize(const char *buffer, int len, int initial_size, int
*outlen);
516     STBIDEF char *stbi_zlib_decode_malloc_guesssize_headerflag(const char *buffer, int len, int
initial_size, int *outlen, int parse_header);
517     STBIDEF char *stbi_zlib_decode_malloc(const char *buffer, int len, int *outlen);
518     STBIDEF int  stbi_zlib_decode_buffer(char *obuffer, int olen, const char *ibuffer, int ilen);
519
520     STBIDEF char *stbi_zlib_decode_noheader_malloc(const char *buffer, int len, int *outlen);
521     STBIDEF int  stbi_zlib_decode_noheader_buffer(char *obuffer, int olen, const char *ibuffer, int
ilen);
522
523
524 #ifdef __cplusplus
525 }
526 #endif
527
528 //
529 //
530 #endif // STBI_INCLUDE_STB_IMAGE_H
531
532
533 #ifdef STB_IMAGE_IMPLEMENTATION
534
535 #if defined(STBI_ONLY_JPEG) || defined(STBI_ONLY_PNG) || defined(STBI_ONLY_BMP) \
536 || defined(STBI_ONLY_TGA) || defined(STBI_ONLY_GIF) || defined(STBI_ONLY_PSD) \
537 || defined(STBI_ONLY_HDR) || defined(STBI_ONLY_PIC) || defined(STBI_ONLY_PNM) \
538 || defined(STBI_ONLY_ZLIB)
539 #ifndef STBI_ONLY_JPEG
540 #define STBI_NO_JPEG
541 #endif
542 #ifndef STBI_ONLY_PNG
543 #define STBI_NO_PNG
544 #endif
545 #ifndef STBI_ONLY_BMP
546 #define STBI_NO_BMP
547 #endif
548 #ifndef STBI_ONLY_PSD
549 #define STBI_NO_PSD
550 #endif
551 #ifndef STBI_ONLY_TGA
552 #define STBI_NO_TGA
553 #endif
554 #ifndef STBI_ONLY_GIF
555 #define STBI_NO_GIF
556 #endif
557 #ifndef STBI_ONLY_HDR
558 #define STBI_NO_HDR
559 #endif
560 #ifndef STBI_ONLY_PIC
561 #define STBI_NO_PIC
562 #endif
563 #ifndef STBI_ONLY_PNM
564 #define STBI_NO_PNM
565 #endif
566 #endif
567
568 #if defined(STBI_NO_PNG) && !defined(STBI_SUPPORT_ZLIB) && !defined(STBI_NO_ZLIB)
569 #define STBI_NO_ZLIB
570 #endif
571
572
573 #include <stdarg.h>
574 #include <stddef.h> // ptrdiff_t on osx
575 #include <stdlib.h>
576 #include <string.h>
577 #include <limits.h>

```

```

578
579 #if !defined(STBI_NO_LINEAR) || !defined(STBI_NO_HDR)
580 #include <math.h> // ldexp
581 #endif
582
583 #ifndef STBI_NO_STDIO
584 #include <stdio.h>
585 #endif
586
587 #ifndef STBI_ASSERT
588 #include <assert.h>
589 #define STBI_ASSERT(x) assert(x)
590 #endif
591
592
593 #ifndef _MSC_VER
594 #ifdef __cplusplus
595 #define stbi_inline inline
596 #else
597 #define stbi_inline
598 #endif
599 #else
600 #define stbi_inline __forceinline
601 #endif
602
603
604 #ifdef _MSC_VER
605 typedef unsigned short stbi__uint16;
606 typedef signed short stbi__int16;
607 typedef unsigned int stbi__uint32;
608 typedef signed int stbi__int32;
609 #else
610 #include <stdint.h>
611 typedef uint16_t stbi__uint16;
612 typedef int16_t stbi__int16;
613 typedef uint32_t stbi__uint32;
614 typedef int32_t stbi__int32;
615 #endif
616
617 // should produce compiler error if size is wrong
618 typedef unsigned char validate_uint32[sizeof(stbi__uint32) == 4 ? 1 : -1];
619
620 #ifdef _MSC_VER
621 #define STBI_NOTUSED(v) (void)(v)
622 #else
623 #define STBI_NOTUSED(v) (void)sizeof(v)
624 #endif
625
626 #ifdef _MSC_VER
627 #define STBI_HAS_LROTL
628 #endif
629
630 #ifdef STBI_HAS_LROTL
631 #define stbi_lrot(x,y) _lrotl(x,y)
632 #else
633 #define stbi_lrot(x,y) (((x) < (y)) | ((x) > (32 - (y))))
634 #endif
635
636 #if defined(STBI_MALLOC) && defined(STBI_FREE) && (defined(STBI_REALLOC) || defined(STBI_REALLOC_SIZED))
637 // ok
638 #elif !defined(STBI_MALLOC) && !defined(STBI_FREE) && !defined(STBI_REALLOC) &&
        !defined(STBI_REALLOC_SIZED)
639 // ok
640 #else
641 #error "Must define all or none of STBI_MALLOC, STBI_FREE, and STBI_REALLOC (or STBI_REALLOC_SIZED)."
642 #endif
643
644 #ifndef STBI_MALLOC
645 #define STBI_MALLOC(sz) malloc(sz)
646 #define STBI_REALLOC(p,newsz) realloc(p,newsz)
647 #define STBI_FREE(p) free(p)
648 #endif
649
650 #ifndef STBI_REALLOC_SIZED
651 #define STBI_REALLOC_SIZED(p,oldsz,newsz) STBI_REALLOC(p,newsz)
652 #endif
653
654 // x86/x64 detection
655 #if defined(__x86_64__) || defined(_M_X64)
656 #define STBI__X64_TARGET
657 #elif defined(__i386__) || defined(_M_I86)
658 #define STBI__X86_TARGET
659 #endif
660
661 #if defined(__GNUC__) && (defined(STBI__X86_TARGET) || defined(STBI__X64_TARGET)) && !defined(__SSE2__)
        && !defined(STBI_NO_SIMD)
662 // NOTE: not clear do we actually need this for the 64-bit path?

```

```

663 // gcc doesn't support sse2 intrinsics unless you compile with -msse2,
664 // (but compiling with -msse2 allows the compiler to use SSE2 everywhere;
665 // this is just broken and gcc are jerks for not fixing it properly
666 // http://www.virtualdub.org/blog/pivot/entry.php?id=363 )
667 #define STBI_NO_SIMD
668 #endif
669
670 #if defined(__MINGW32__) && defined(STBI__X86_TARGET) && !defined(STBI_MINGW_ENABLE_SSE2) &&
    !defined(STBI_NO_SIMD)
671 // Note that __MINGW32__ doesn't actually mean 32-bit, so we have to avoid STBI__X64_TARGET
672 //
673 // 32-bit MinGW wants ESP to be 16-byte aligned, but this is not in the
674 // Windows ABI and VC++ as well as Windows DLLs don't maintain that invariant.
675 // As a result, enabling SSE2 on 32-bit MinGW is dangerous when not
676 // simultaneously enabling "-mstackrealign".
677 //
678 // See https://github.com/nothings/stb/issues/81 for more information.
679 //
680 // So default to no SSE2 on 32-bit MinGW. If you've read this far and added
681 // -mstackrealign to your build settings, feel free to #define STBI_MINGW_ENABLE_SSE2.
682 #define STBI_NO_SIMD
683 #endif
684
685 #if !defined(STBI_NO_SIMD) && (defined(STBI__X86_TARGET) || defined(STBI__X64_TARGET))
686 #define STBI_SSE2
687 #include <emmintrin.h>
688
689 #ifdef _MSC_VER
690
691 #if _MSC_VER >= 1400 // not VC6
692 #include <intrin.h> // __cpuid
693 static int stbi__cpuid3(void)
694 {
695     int info[4];
696     __cpuid(info, 1);
697     return info[3];
698 }
699 #else
700 static int stbi__cpuid3(void)
701 {
702     int res;
703     __asm {
704         mov eax, 1
705         cpuid
706         mov res, edx
707     }
708     return res;
709 }
710 #endif
711
712 #define STBI_SIMD_ALIGN(type, name) __declspec(aligned(16)) type name
713
714 static int stbi__sse2_available()
715 {
716     int info3 = stbi__cpuid3();
717     return ((info3 >> 26) & 1) != 0;
718 }
719 #else // assume GCC-style if not VC++
720 #define STBI_SIMD_ALIGN(type, name) type name __attribute__((aligned(16)))
721
722 static int stbi__sse2_available()
723 {
724     #if defined(__GNUC__) && (__GNUC__ * 100 + __GNUC_MINOR__) >= 408 // GCC 4.8 or later
725         // GCC 4.8+ has a nice way to do this
726         return __builtin_cpu_supports("sse2");
727     #else
728         // portable way to do this, preferably without using GCC inline ASM?
729         // just bail for now.
730         return 0;
731     #endif
732 }
733 #endif
734 #endif
735
736 // ARM NEON
737 #if defined(STBI_NO_SIMD) && defined(STBI_NEON)
738 #undef STBI_NEON
739 #endif
740
741 #ifdef STBI_NEON
742 #include <arm_neon.h>
743 // assume GCC or Clang on ARM targets
744 #define STBI_SIMD_ALIGN(type, name) type name __attribute__((aligned(16)))
745 #endif
746
747 #ifndef STBI_SIMD_ALIGN
748 #define STBI_SIMD_ALIGN(type, name) type name

```

```

749 #endif
750
752 //
753 // stbi__context struct and start_xxx functions
754
755 // stbi__context structure is our basic context used by all images, so it
756 // contains all the IO context, plus some basic image information
757 typedef struct
758 {
759     stbi__uint32 img_x, img_y;
760     int img_n, img_out_n;
761
762     stbi_io_callbacks io;
763     void *io_user_data;
764
765     int read_from_callbacks;
766     int buflen;
767     stbi_uc buffer_start[128];
768
769     stbi_uc *img_buffer, *img_buffer_end;
770     stbi_uc *img_buffer_original, *img_buffer_original_end;
771 } stbi__context;
772
773
774 static void stbi__refill_buffer(stbi__context *s);
775
776 // initialize a memory-decode context
777 static void stbi__start_mem(stbi__context *s, stbi_uc const *buffer, int len)
778 {
779     s->io.read = NULL;
780     s->read_from_callbacks = 0;
781     s->img_buffer = s->img_buffer_original = (stbi_uc *)buffer;
782     s->img_buffer_end = s->img_buffer_original_end = (stbi_uc *)buffer + len;
783 }
784
785 // initialize a callback-based context
786 static void stbi__start_callbacks(stbi__context *s, stbi_io_callbacks *c, void *user)
787 {
788     s->io = *c;
789     s->io_user_data = user;
790     s->buflen = sizeof(s->buffer_start);
791     s->read_from_callbacks = 1;
792     s->img_buffer_original = s->buffer_start;
793     stbi__refill_buffer(s);
794     s->img_buffer_original_end = s->img_buffer_end;
795 }
796
797 #ifndef STBI_NO_STDIO
798
799 static int stbi__stdio_read(void *user, char *data, int size)
800 {
801     return (int)fread(data, 1, size, (FILE*)user);
802 }
803
804 static void stbi__stdio_skip(void *user, int n)
805 {
806     fseek((FILE*)user, n, SEEK_CUR);
807 }
808
809 static int stbi__stdio_eof(void *user)
810 {
811     return feof((FILE*)user);
812 }
813
814 static stbi_io_callbacks stbi__stdio_callbacks =
815 {
816     stbi__stdio_read,
817     stbi__stdio_skip,
818     stbi__stdio_eof,
819 };
820
821 static void stbi__start_file(stbi__context *s, FILE *f)
822 {
823     stbi__start_callbacks(s, &stbi__stdio_callbacks, (void *)f);
824 }
825
826 //static void stop_file(stbi__context *s) { }
827
828 #endif // !STBI_NO_STDIO
829
830 static void stbi__rewind(stbi__context *s)
831 {
832     // conceptually rewind SHOULD rewind to the beginning of the stream,
833     // but we just rewind to the beginning of the initial buffer, because
834     // we only use it after doing 'test', which only ever looks at at most 92 bytes
835     s->img_buffer = s->img_buffer_original;
836     s->img_buffer_end = s->img_buffer_original_end;

```

```

837 }
838
839 enum
840 {
841     STBI_ORDER_RGB,
842     STBI_ORDER_BGR
843 };
844
845 typedef struct
846 {
847     int bits_per_channel;
848     int num_channels;
849     int channel_order;
850 } stbi__result_info;
851
852 #ifndef STBI_NO_JPEG
853 static int stbi__jpeg_test(stbi__context *s);
854 static void stbi__jpeg_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
855     stbi__result_info *ri);
856 static int stbi__jpeg_info(stbi__context *s, int *x, int *y, int *comp);
857 #endif
858
859 #ifndef STBI_NO_PNG
860 static int stbi__png_test(stbi__context *s);
861 static void stbi__png_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
862     stbi__result_info *ri);
863 static int stbi__png_info(stbi__context *s, int *x, int *y, int *comp);
864 #endif
865
866 #ifndef STBI_NO_BMP
867 static int stbi__bmp_test(stbi__context *s);
868 static void stbi__bmp_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
869     stbi__result_info *ri);
870 static int stbi__bmp_info(stbi__context *s, int *x, int *y, int *comp);
871 #endif
872
873 #ifndef STBI_NO_TGA
874 static int stbi__tga_test(stbi__context *s);
875 static void stbi__tga_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
876     stbi__result_info *ri);
877 static int stbi__tga_info(stbi__context *s, int *x, int *y, int *comp);
878 #endif
879
880 #ifndef STBI_NO_PSD
881 static int stbi__psd_test(stbi__context *s);
882 static void stbi__psd_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
883     stbi__result_info *ri, int bpc);
884 static int stbi__psd_info(stbi__context *s, int *x, int *y, int *comp);
885 #endif
886
887 #ifndef STBI_NO_HDR
888 static int stbi__hdr_test(stbi__context *s);
889 static float stbi__hdr_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
890     stbi__result_info *ri);
891 static int stbi__hdr_info(stbi__context *s, int *x, int *y, int *comp);
892 #endif
893
894 #ifndef STBI_NO_PIC
895 static int stbi__pic_test(stbi__context *s);
896 static void stbi__pic_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
897     stbi__result_info *ri);
898 static int stbi__pic_info(stbi__context *s, int *x, int *y, int *comp);
899 #endif
900
901 #ifndef STBI_NO_GIF
902 static int stbi__gif_test(stbi__context *s);
903 static void stbi__gif_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
904     stbi__result_info *ri);
905 static int stbi__gif_info(stbi__context *s, int *x, int *y, int *comp);
906 #endif
907
908 #ifndef STBI_NO_PNM
909 static int stbi__pnm_test(stbi__context *s);
910 static void stbi__pnm_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
911     stbi__result_info *ri);
912 static int stbi__pnm_info(stbi__context *s, int *x, int *y, int *comp);
913 #endif
914
915 // this is not threadsafe
916 static const char *stbi__g_failure_reason;
917
918 STBIDEF const char *stbi_failure_reason(void)
919 {
920     return stbi__g_failure_reason;
921 }
922
923 static int stbi__err(const char *str)

```

```

915 {
916     stbi__g_failure_reason = str;
917     return 0;
918 }
919
920 static void *stbi__malloc(size_t size)
921 {
922     return STBI_MALLOC(size);
923 }
924
925 // stb_image uses ints pervasively, including for offset calculations.
926 // therefore the largest decoded image size we can support with the
927 // current code, even on 64-bit targets, is INT_MAX. this is not a
928 // significant limitation for the intended use case.
929 //
930 // we do, however, need to make sure our size calculations don't
931 // overflow. hence a few helper functions for size calculations that
932 // multiply integers together, making sure that they're non-negative
933 // and no overflow occurs.
934
935 // return 1 if the sum is valid, 0 on overflow.
936 // negative terms are considered invalid.
937 static int stbi__addsizes_valid(int a, int b)
938 {
939     if (b < 0) return 0;
940     // now 0 <= b <= INT_MAX, hence also
941     // 0 <= INT_MAX - b <= INTMAX.
942     // And "a + b <= INT_MAX" (which might overflow) is the
943     // same as a <= INT_MAX - b (no overflow)
944     return a <= INT_MAX - b;
945 }
946
947 // returns 1 if the product is valid, 0 on overflow.
948 // negative factors are considered invalid.
949 static int stbi__mul2sizes_valid(int a, int b)
950 {
951     if (a < 0 || b < 0) return 0;
952     if (b == 0) return 1; // mul-by-0 is always safe
953     // portable way to check for no overflows in a*b
954     return a <= INT_MAX / b;
955 }
956
957 // returns 1 if "a*b + add" has no negative terms/factors and doesn't overflow
958 static int stbi__mad2sizes_valid(int a, int b, int add)
959 {
960     return stbi__mul2sizes_valid(a, b) && stbi__addsizes_valid(a*b, add);
961 }
962
963 // returns 1 if "a*b*c + add" has no negative terms/factors and doesn't overflow
964 static int stbi__mad3sizes_valid(int a, int b, int c, int add)
965 {
966     return stbi__mul2sizes_valid(a, b) && stbi__mul2sizes_valid(a*b, c) &&
967         stbi__addsizes_valid(a*b*c, add);
968 }
969
970 // returns 1 if "a*b*c*d + add" has no negative terms/factors and doesn't overflow
971 static int stbi__mad4sizes_valid(int a, int b, int c, int d, int add)
972 {
973     return stbi__mul2sizes_valid(a, b) && stbi__mul2sizes_valid(a*b, c) &&
974         stbi__mul2sizes_valid(a*b*c, d) && stbi__addsizes_valid(a*b*c*d, add);
975 }
976
977 // mallocs with size overflow checking
978 static void *stbi__malloc_mad2(int a, int b, int add)
979 {
980     if (!stbi__mad2sizes_valid(a, b, add)) return NULL;
981     return stbi__malloc(a*b + add);
982 }
983
984 static void *stbi__malloc_mad3(int a, int b, int c, int add)
985 {
986     if (!stbi__mad3sizes_valid(a, b, c, add)) return NULL;
987     return stbi__malloc(a*b*c + add);
988 }
989
990 static void *stbi__malloc_mad4(int a, int b, int c, int d, int add)
991 {
992     if (!stbi__mad4sizes_valid(a, b, c, d, add)) return NULL;
993     return stbi__malloc(a*b*c*d + add);
994 }
995
996 // stbi__err - error
997 // stbi__errpf - error returning pointer to float
998 // stbi__errpuc - error returning pointer to unsigned char
999
1000 #ifdef STBI_NO_FAILURE_STRINGS
1001 #define stbi__err(x,y) 0

```

```

1002 #elif defined(STBI_FAILURE_USERMSG)
1003 #define stbi__err(x,y) stbi__err(y)
1004 #else
1005 #define stbi__err(x,y) stbi__err(x)
1006 #endif
1007
1008 #define stbi__errpf(x,y) ((float *) (size_t) (stbi__err(x,y)?NULL:NULL))
1009 #define stbi__errpuc(x,y) ((unsigned char *) (size_t) (stbi__err(x,y)?NULL:NULL))
1010
1011 STBIDEF void stbi_image_free(void *retval_from_stbi_load)
1012 {
1013     STBI_FREE(retval_from_stbi_load);
1014 }
1015
1016 #ifndef STBI_NO_LINEAR
1017 static float *stbi_ldr_to_hdr(stbi_uc *data, int x, int y, int comp);
1018 #endif
1019
1020 #ifndef STBI_NO_HDR
1021 static stbi_uc *stbi_hdr_to_ldr(float *data, int x, int y, int comp);
1022 #endif
1023
1024 static int stbi__vertically_flip_on_load = 0;
1025
1026 STBIDEF void stbi_set_flip_vertically_on_load(int flag_true_if_should_flip)
1027 {
1028     stbi__vertically_flip_on_load = flag_true_if_should_flip;
1029 }
1030
1031 static void stbi__load_main(stbi__context *s, int *x, int *y, int *comp, int req_comp,
1032     stbi__result_info *ri, int bpc)
1033 {
1034     memset(ri, 0, sizeof(*ri)); // make sure it's initialized if we add new fields
1035     ri->bits_per_channel = 8; // default is 8 so most paths don't have to be changed
1036     ri->channel_order = STBI_ORDER_RGB; // all current input & output are this, but this is here so we
1037     // can add BGR order
1038     ri->num_channels = 0;
1039
1040 #ifndef STBI_NO_JPEG
1041     if (stbi__jpeg_test(s)) return stbi__jpeg_load(s, x, y, comp, req_comp, ri);
1042 #endif
1043 #ifndef STBI_NO_PNG
1044     if (stbi__png_test(s)) return stbi__png_load(s, x, y, comp, req_comp, ri);
1045 #endif
1046 #ifndef STBI_NO_BMP
1047     if (stbi__bmp_test(s)) return stbi__bmp_load(s, x, y, comp, req_comp, ri);
1048 #endif
1049 #ifndef STBI_NO_GIF
1050     if (stbi__gif_test(s)) return stbi__gif_load(s, x, y, comp, req_comp, ri);
1051 #endif
1052 #ifndef STBI_NO_PSD
1053     if (stbi__psd_test(s)) return stbi__psd_load(s, x, y, comp, req_comp, ri, bpc);
1054 #endif
1055 #ifndef STBI_NO_PIC
1056     if (stbi__pic_test(s)) return stbi__pic_load(s, x, y, comp, req_comp, ri);
1057 #endif
1058 #ifndef STBI_NO_PNM
1059     if (stbi__pnm_test(s)) return stbi__pnm_load(s, x, y, comp, req_comp, ri);
1060 #endif
1061
1062 #ifndef STBI_NO_HDR
1063     if (stbi__hdr_test(s)) {
1064         float *hdr = stbi__hdr_load(s, x, y, comp, req_comp, ri);
1065         return stbi__hdr_to_ldr(hdr, *x, *y, req_comp ? req_comp : *comp);
1066     }
1067 #endif
1068
1069 #ifndef STBI_NO_TGA
1070     // test tga last because it's a crappy test!
1071     if (stbi__tga_test(s))
1072         return stbi__tga_load(s, x, y, comp, req_comp, ri);
1073 #endif
1074
1075     return stbi__errpuc("unknown image type", "Image not of any known type, or corrupt");
1076 }
1077
1078 static stbi_uc *stbi_convert_16_to_8(stbi_uint16 *orig, int w, int h, int channels)
1079 {
1080     int i;
1081     int img_len = w * h * channels;
1082     stbi_uc *reduced;
1083
1084     reduced = (stbi_uc *) stbi__malloc(img_len);
1085     if (reduced == NULL) return stbi__errpuc("outofmem", "Out of memory");
1086
1087     for (i = 0; i < img_len; ++i)
1088         reduced[i] = (stbi_uc) ((orig[i] >> 8) & 0xFF); // top half of each byte is sufficient approx of

```



```

16->8 bit scaling
1087
1088     STBI_FREE(orig);
1089     return reduced;
1090 }
1091
1092 static stbi_uint16 *stbi_convert_8_to_16(stbi_uc *orig, int w, int h, int channels)
1093 {
1094     int i;
1095     int img_len = w * h * channels;
1096     stbi_uint16 *enlarged;
1097
1098     enlarged = (stbi_uint16 *)stbi_malloc(img_len * 2);
1099     if (enlarged == NULL) return (stbi_uint16 *)stbi_errpuc("outofmem", "Out of memory");
1100
1101     for (i = 0; i < img_len; ++i)
1102         enlarged[i] = (stbi_uint16)((orig[i] < 8) + orig[i]); // replicate to high and low byte, maps
1103         0->0, 255->0xffff
1104
1105     STBI_FREE(orig);
1106     return enlarged;
1107 }
1108
1109 static unsigned char *stbi_load_and_postprocess_8bit(stbi_context *s, int *x, int *y, int *comp, int
    req_comp)
1110 {
1111     stbi_result_info ri;
1112     void *result = stbi_load_main(s, x, y, comp, req_comp, &ri, 8);
1113
1114     if (result == NULL)
1115         return NULL;
1116
1117     if (ri.bits_per_channel != 8) {
1118         STBI_ASSERT(ri.bits_per_channel == 16);
1119         result = stbi_convert_16_to_8((stbi_uint16 *)result, *x, *y, req_comp == 0 ? *comp :
    req_comp);
1120         ri.bits_per_channel = 8;
1121     }
1122
1123     // @TODO: move stbi_convert_format to here
1124
1125     if (stbi_vertically_flip_on_load) {
1126         int w = *x, h = *y;
1127         int channels = req_comp ? req_comp : *comp;
1128         int row, col, z;
1129         stbi_uc *image = (stbi_uc *)result;
1130
1131         // @OPTIMIZE: use a bigger temp buffer and memcpy multiple pixels at once
1132         for (row = 0; row < (h >> 1); row++) {
1133             for (col = 0; col < w; col++) {
1134                 for (z = 0; z < channels; z++) {
1135                     stbi_uc temp = image[(row * w + col) * channels + z];
1136                     image[(row * w + col) * channels + z] = image[((h - row - 1) * w + col) * channels
    + z];
1137                     image[((h - row - 1) * w + col) * channels + z] = temp;
1138                 }
1139             }
1140         }
1141     }
1142
1143     return (unsigned char *)result;
1144 }
1145
1146 static stbi_uint16 *stbi_load_and_postprocess_16bit(stbi_context *s, int *x, int *y, int *comp, int
    req_comp)
1147 {
1148     stbi_result_info ri;
1149     void *result = stbi_load_main(s, x, y, comp, req_comp, &ri, 16);
1150
1151     if (result == NULL)
1152         return NULL;
1153
1154     if (ri.bits_per_channel != 16) {
1155         STBI_ASSERT(ri.bits_per_channel == 8);
1156         result = stbi_convert_8_to_16((stbi_uc *)result, *x, *y, req_comp == 0 ? *comp : req_comp);
1157         ri.bits_per_channel = 16;
1158     }
1159
1160     // @TODO: move stbi_convert_format16 to here
1161     // @TODO: special case RGB-to-Y (and RGBA-to-YA) for 8-bit-to-16-bit case to keep more precision
1162
1163     if (stbi_vertically_flip_on_load) {
1164         int w = *x, h = *y;
1165         int channels = req_comp ? req_comp : *comp;
1166         int row, col, z;
1167         stbi_uint16 *image = (stbi_uint16 *)result;
1168
1169         for (row = 0; row < (h >> 1); row++) {
1170             for (col = 0; col < w; col++) {
1171                 for (z = 0; z < channels; z++) {
1172                     stbi_uint16 temp = image[(row * w + col) * channels + z];
1173                     image[(row * w + col) * channels + z] = image[((h - row - 1) * w + col) * channels
    + z];
1174                     image[((h - row - 1) * w + col) * channels + z] = temp;
1175                 }
1176             }
1177         }
1178     }
1179
1180     return image;
1181 }

```

```

1168         // @OPTIMIZE: use a bigger temp buffer and memcpy multiple pixels at once
1169         for (row = 0; row < (h >> 1); row++) {
1170             for (col = 0; col < w; col++) {
1171                 for (z = 0; z < channels; z++) {
1172                     stbi__uint16 temp = image[(row * w + col) * channels + z];
1173                     image[(row * w + col) * channels + z] = image[((h - row - 1) * w + col) * channels
+ z];
1174                     image[((h - row - 1) * w + col) * channels + z] = temp;
1175                 }
1176             }
1177         }
1178     }
1179     return (stbi__uint16 *)result;
1180 }
1181
1182 #ifndef STBI_NO_HDR
1183 static void stbi__float_postprocess(float *result, int *x, int *y, int *comp, int req_comp)
1184 {
1185     if (stbi__vertically_flip_on_load && result != NULL) {
1186         int w = *x, h = *y;
1187         int depth = req_comp ? req_comp : *comp;
1188         int row, col, z;
1189         float temp;
1190
1191         // @OPTIMIZE: use a bigger temp buffer and memcpy multiple pixels at once
1192         for (row = 0; row < (h >> 1); row++) {
1193             for (col = 0; col < w; col++) {
1194                 for (z = 0; z < depth; z++) {
1195                     temp = result[(row * w + col) * depth + z];
1196                     result[(row * w + col) * depth + z] = result[((h - row - 1) * w + col) * depth +
z];
1197                     result[((h - row - 1) * w + col) * depth + z] = temp;
1198                 }
1199             }
1200         }
1201     }
1202 }
1203
1204 #endif
1205
1206 #ifndef STBI_NO_STDIO
1207 static FILE *stbi__fopen(char const *filename, char const *mode)
1208 {
1209     FILE *f;
1210     #if defined(_MSC_VER) && _MSC_VER >= 1400
1211     if (0 != fopen_s(&f, filename, mode))
1212         f = 0;
1213     #else
1214     f = fopen(filename, mode);
1215     #endif
1216     return f;
1217 }
1218
1219 STBIDEF stbi_uc *stbi_load(char const *filename, int *x, int *y, int *comp, int req_comp)
1220 {
1221     FILE *f = stbi__fopen(filename, "rb");
1222     unsigned char *result;
1223     if (!f) return stbi__errpuc("can't fopen", "Unable to open file");
1224     result = stbi_load_from_file(f, x, y, comp, req_comp);
1225     fclose(f);
1226     return result;
1227 }
1228
1229 STBIDEF stbi_uc *stbi_load_from_file(FILE *f, int *x, int *y, int *comp, int req_comp)
1230 {
1231     unsigned char *result;
1232     stbi__context s;
1233     stbi__start_file(&s, f);
1234     result = stbi_load_and_postprocess_8bit(&s, x, y, comp, req_comp);
1235     if (result) {
1236         // need to 'unget' all the characters in the IO buffer
1237         fseek(f, -(int)(s.img_buffer_end - s.img_buffer), SEEK_CUR);
1238     }
1239     return result;
1240 }
1241
1242 STBIDEF stbi__uint16 *stbi_load_from_file_16(FILE *f, int *x, int *y, int *comp, int req_comp)
1243 {
1244     stbi__uint16 *result;
1245     stbi__context s;
1246     stbi__start_file(&s, f);
1247     result = stbi_load_and_postprocess_16bit(&s, x, y, comp, req_comp);
1248     if (result) {
1249         // need to 'unget' all the characters in the IO buffer
1250         fseek(f, -(int)(s.img_buffer_end - s.img_buffer), SEEK_CUR);
1251     }
1252     return result;
1253 }

```

```

1253     }
1254     return result;
1255 }
1256
1257 STBIDEF stbi_us *stbi_load_16(char const *filename, int *x, int *y, int *comp, int req_comp)
1258 {
1259     FILE *f = stbi__fopen(filename, "rb");
1260     stbi__uint16 *result;
1261     if (!f) return (stbi_us *)stbi__errpuc("can't fopen", "Unable to open file");
1262     result = stbi_load_from_file_16(f, x, y, comp, req_comp);
1263     fclose(f);
1264     return result;
1265 }
1266
1267
1268 #endif
1269
1270 STBIDEF stbi_uc *stbi_load_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp, int
    req_comp)
1271 {
1272     stbi__context s;
1273     stbi__start_mem(&s, buffer, len);
1274     return stbi__load_and_postprocess_8bit(&s, x, y, comp, req_comp);
1275 }
1276
1277 STBIDEF stbi_uc *stbi_load_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y,
    int *comp, int req_comp)
1278 {
1279     stbi__context s;
1280     stbi__start_callbacks(&s, (stbi_io_callbacks *)clbk, user);
1281     return stbi__load_and_postprocess_8bit(&s, x, y, comp, req_comp);
1282 }
1283
1284 #ifndef STBI_NO_LINEAR
1285 static float *stbi__loadf_main(stbi__context *s, int *x, int *y, int *comp, int req_comp)
1286 {
1287     unsigned char *data;
1288     #ifndef STBI_NO_HDR
1289     if (stbi__hdr_test(s)) {
1290         stbi__result_info ri;
1291         float *hdr_data = stbi__hdr_load(s, x, y, comp, req_comp, &ri);
1292         if (hdr_data)
1293             stbi__float_postprocess(hdr_data, x, y, comp, req_comp);
1294         return hdr_data;
1295     }
1296     #endif
1297     data = stbi__load_and_postprocess_8bit(s, x, y, comp, req_comp);
1298     if (data)
1299         return stbi__ldr_to_hdr(data, *x, *y, req_comp ? req_comp : *comp);
1300     return stbi__errpf("unknown image type", "Image not of any known type, or corrupt");
1301 }
1302
1303 STBIDEF float *stbi_loadf_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp, int
    req_comp)
1304 {
1305     stbi__context s;
1306     stbi__start_mem(&s, buffer, len);
1307     return stbi__loadf_main(&s, x, y, comp, req_comp);
1308 }
1309
1310 STBIDEF float *stbi_loadf_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y, int
    *comp, int req_comp)
1311 {
1312     stbi__context s;
1313     stbi__start_callbacks(&s, (stbi_io_callbacks *)clbk, user);
1314     return stbi__loadf_main(&s, x, y, comp, req_comp);
1315 }
1316
1317 #ifndef STBI_NO_STDIO
1318 STBIDEF float *stbi_loadf(char const *filename, int *x, int *y, int *comp, int req_comp)
1319 {
1320     float *result;
1321     FILE *f = stbi__fopen(filename, "rb");
1322     if (!f) return stbi__errpf("can't fopen", "Unable to open file");
1323     result = stbi_loadf_from_file(f, x, y, comp, req_comp);
1324     fclose(f);
1325     return result;
1326 }
1327
1328 STBIDEF float *stbi_loadf_from_file(FILE *f, int *x, int *y, int *comp, int req_comp)
1329 {
1330     stbi__context s;
1331     stbi__start_file(&s, f);
1332     return stbi__loadf_main(&s, x, y, comp, req_comp);
1333 }
1334 #endif // !STBI_NO_STDIO
1335

```

```

1336 #endif // !STBI_NO_LINEAR
1337
1338 // these is-hdr-or-not is defined independent of whether STBI_NO_LINEAR is
1339 // defined, for API simplicity; if STBI_NO_LINEAR is defined, it always
1340 // reports false!
1341
1342 STBIDEF int stbi_is_hdr_from_memory(stbi_uc const *buffer, int len)
1343 {
1344 #ifndef STBI_NO_HDR
1345     stbi__context s;
1346     stbi__start_mem(&s, buffer, len);
1347     return stbi_hdr_test(&s);
1348 #else
1349     STBI_NOTUSED(buffer);
1350     STBI_NOTUSED(len);
1351     return 0;
1352 #endif
1353 }
1354
1355 #ifndef STBI_NO_STDIO
1356 STBIDEF int stbi_is_hdr(char const *filename)
1357 {
1358     FILE *f = stbi_fopen(filename, "rb");
1359     int result = 0;
1360     if (f) {
1361         result = stbi_is_hdr_from_file(f);
1362         fclose(f);
1363     }
1364     return result;
1365 }
1366
1367 STBIDEF int stbi_is_hdr_from_file(FILE *f)
1368 {
1369 #ifndef STBI_NO_HDR
1370     stbi__context s;
1371     stbi__start_file(&s, f);
1372     return stbi_hdr_test(&s);
1373 #else
1374     STBI_NOTUSED(f);
1375     return 0;
1376 #endif
1377 }
1378 #endif // !STBI_NO_STDIO
1379
1380 STBIDEF int stbi_is_hdr_from_callbacks(stbi_io_callbacks const *clbk, void *user)
1381 {
1382 #ifndef STBI_NO_HDR
1383     stbi__context s;
1384     stbi__start_callbacks(&s, (stbi_io_callbacks *)clbk, user);
1385     return stbi_hdr_test(&s);
1386 #else
1387     STBI_NOTUSED(clbk);
1388     STBI_NOTUSED(user);
1389     return 0;
1390 #endif
1391 }
1392
1393 #ifndef STBI_NO_LINEAR
1394 static float stbi__l2h_gamma = 2.2f, stbi__l2h_scale = 1.0f;
1395
1396 STBIDEF void stbi_ldr_to_hdr_gamma(float gamma) { stbi__l2h_gamma = gamma; }
1397 STBIDEF void stbi_ldr_to_hdr_scale(float scale) { stbi__l2h_scale = scale; }
1398 #endif
1399
1400 static float stbi__h2l_gamma_i = 1.0f / 2.2f, stbi__h2l_scale_i = 1.0f;
1401
1402 STBIDEF void stbi_hdr_to_ldr_gamma(float gamma) { stbi__h2l_gamma_i = 1 / gamma; }
1403 STBIDEF void stbi_hdr_to_ldr_scale(float scale) { stbi__h2l_scale_i = 1 / scale; }
1404
1405
1406 //
1407 // Common code used by all image loaders
1408 //
1409 //
1410
1411 enum
1412 {
1413     STBI__SCAN_load = 0,
1414     STBI__SCAN_type,
1415     STBI__SCAN_header
1416 };
1417
1418 static void stbi__refill_buffer(stbi__context *s)
1419 {
1420     int n = (s->io.read)(s->io_user_data, (char*)s->buffer_start, s->buflen);
1421     if (n == 0) {
1422         // at end of file, treat same as if from memory, but need to handle case
1423         // where s->img_buffer isn't pointing to safe memory, e.g. 0-byte file

```

```

1424         s->read_from_callbacks = 0;
1425         s->img_buffer = s->buffer_start;
1426         s->img_buffer_end = s->buffer_start + 1;
1427         *s->img_buffer = 0;
1428     }
1429     else {
1430         s->img_buffer = s->buffer_start;
1431         s->img_buffer_end = s->buffer_start + n;
1432     }
1433 }
1434
1435 stbi_inline static stbi_uc stbi__get8(stbi__context *s)
1436 {
1437     if (s->img_buffer < s->img_buffer_end)
1438         return *s->img_buffer++;
1439     if (s->read_from_callbacks) {
1440         stbi__refill_buffer(s);
1441         return *s->img_buffer++;
1442     }
1443     return 0;
1444 }
1445
1446 stbi_inline static int stbi__at_eof(stbi__context *s)
1447 {
1448     if (s->io.read) {
1449         if (!(s->io.eof) (s->io_user_data)) return 0;
1450         // if feof() is true, check if buffer = end
1451         // special case: we've only got the special 0 character at the end
1452         if (s->read_from_callbacks == 0) return 1;
1453     }
1454
1455     return s->img_buffer >= s->img_buffer_end;
1456 }
1457
1458 static void stbi__skip(stbi__context *s, int n)
1459 {
1460     if (n < 0) {
1461         s->img_buffer = s->img_buffer_end;
1462         return;
1463     }
1464     if (s->io.read) {
1465         int blen = (int) (s->img_buffer_end - s->img_buffer);
1466         if (blen < n) {
1467             s->img_buffer = s->img_buffer_end;
1468             (s->io.skip) (s->io_user_data, n - blen);
1469             return;
1470         }
1471     }
1472     s->img_buffer += n;
1473 }
1474
1475 static int stbi__getn(stbi__context *s, stbi_uc *buffer, int n)
1476 {
1477     if (s->io.read) {
1478         int blen = (int) (s->img_buffer_end - s->img_buffer);
1479         if (blen < n) {
1480             int res, count;
1481
1482             memcpy(buffer, s->img_buffer, blen);
1483
1484             count = (s->io.read) (s->io_user_data, (char*)buffer + blen, n - blen);
1485             res = (count == (n - blen));
1486             s->img_buffer = s->img_buffer_end;
1487             return res;
1488         }
1489     }
1490
1491     if (s->img_buffer + n <= s->img_buffer_end) {
1492         memcpy(buffer, s->img_buffer, n);
1493         s->img_buffer += n;
1494         return 1;
1495     }
1496     else
1497         return 0;
1498 }
1499
1500 static int stbi__get16be(stbi__context *s)
1501 {
1502     int z = stbi__get8(s);
1503     return (z << 8) + stbi__get8(s);
1504 }
1505
1506 static stbi_uint32 stbi__get32be(stbi__context *s)
1507 {
1508     stbi_uint32 z = stbi__get16be(s);
1509     return (z << 16) + stbi__get16be(s);
1510 }

```

```

1511
1512 #if defined(STBI_NO_BMP) && defined(STBI_NO_TGA) && defined(STBI_NO_GIF)
1513 // nothing
1514 #else
1515 static int stbi_get16le(stbi__context *s)
1516 {
1517     int z = stbi_get8(s);
1518     return z + (stbi_get8(s) << 8);
1519 }
1520 #endif
1521
1522 #ifndef STBI_NO_BMP
1523 static stbi_uint32 stbi_get32le(stbi__context *s)
1524 {
1525     stbi_uint32 z = stbi_get16le(s);
1526     return z + (stbi_get16le(s) << 16);
1527 }
1528 #endif
1529
1530 #define STBI__BYTECAST(x) ((stbi_uc) ((x) & 255)) // truncate int to byte without warnings
1531
1532
1533 //
1534 // generic converter from built-in img_n to req_comp
1535 // individual types do this automatically as much as possible (e.g. jpeg
1536 // does all cases internally since it needs to colorspace convert anyway,
1537 // and it never has alpha, so very few cases). png can automatically
1538 // interleave an alpha=255 channel, but falls back to this for other cases
1539 //
1540 // assume data buffer is malloced, so malloc a new one and free that one
1541 // only failure mode is malloc failing
1542
1543 static stbi_uc stbi__compute_y(int r, int g, int b)
1544 {
1545     return (stbi_uc) (((r * 77) + (g * 150) + (29 * b)) >> 8);
1546 }
1547
1548
1549 static unsigned char *stbi__convert_format(unsigned char *data, int img_n, int req_comp, unsigned int
x, unsigned int y)
1550 {
1551     int i, j;
1552     unsigned char *good;
1553
1554     if (req_comp == img_n) return data;
1555     STBI_ASSERT(req_comp >= 1 && req_comp <= 4);
1556
1557     good = (unsigned char *)stbi__malloc_mad3(req_comp, x, y, 0);
1558     if (good == NULL) {
1559         STBI_FREE(data);
1560         return stbi__errpuc("outofmem", "Out of memory");
1561     }
1562
1563     for (j = 0; j < (int)y; ++j) {
1564         unsigned char *src = data + j * x * img_n;
1565         unsigned char *dest = good + j * x * req_comp;
1566
1567 #define STBI__COMBO(a,b) ((a)*8+(b))
1568 #define STBI__CASE(a,b) case STBI__COMBO(a,b): for(i=x-1; i >= 0; --i, src += a, dest += b)
1569 // convert source image with img_n components to one with req_comp components;
1570 // avoid switch per pixel, so use switch per scanline and massive macros
1571 switch (STBI__COMBO(img_n, req_comp)) {
1572     STBI__CASE(1, 2) { dest[0] = src[0], dest[1] = 255; } break;
1573     STBI__CASE(1, 3) { dest[0] = dest[1] = dest[2] = src[0]; } break;
1574     STBI__CASE(1, 4) { dest[0] = dest[1] = dest[2] = src[0], dest[3] = 255; } break;
1575     STBI__CASE(2, 1) { dest[0] = src[0]; } break;
1576     STBI__CASE(2, 3) { dest[0] = dest[1] = dest[2] = src[0]; } break;
1577     STBI__CASE(2, 4) { dest[0] = dest[1] = dest[2] = src[0], dest[3] = src[1]; } break;
1578     STBI__CASE(3, 4) { dest[0] = src[0], dest[1] = src[1], dest[2] = src[2], dest[3] = 255; }
break;
1579     STBI__CASE(3, 1) { dest[0] = stbi__compute_y(src[0], src[1], src[2]); } break;
1580     STBI__CASE(3, 2) { dest[0] = stbi__compute_y(src[0], src[1], src[2]), dest[1] = 255; }
break;
1581     STBI__CASE(4, 1) { dest[0] = stbi__compute_y(src[0], src[1], src[2]); } break;
1582     STBI__CASE(4, 2) { dest[0] = stbi__compute_y(src[0], src[1], src[2]), dest[1] = src[3]; }
break;
1583     STBI__CASE(4, 3) { dest[0] = src[0], dest[1] = src[1], dest[2] = src[2]; } break;
1584     default: STBI_ASSERT(0);
1585 }
1586 #undef STBI__CASE
1587 }
1588
1589 STBI_FREE(data);
1590 return good;
1591 }
1592
1593 static stbi_uint16 stbi__compute_y_16(int r, int g, int b)
1594 {

```

```

1595     return (stbi_uint16)((r * 77) + (g * 150) + (29 * b)) >> 8);
1596 }
1597
1598 static stbi_uint16 *stbi_convert_format16(stbi_uint16 *data, int img_n, int req_comp, unsigned int
1599     x, unsigned int y)
1600 {
1601     int i, j;
1602     stbi_uint16 *good;
1603
1604     if (req_comp == img_n) return data;
1605     STBI_ASSERT(req_comp >= 1 && req_comp <= 4);
1606
1607     good = (stbi_uint16 *)stbi_malloc(req_comp * x * y * 2);
1608     if (good == NULL) {
1609         STBI_FREE(data);
1610         return (stbi_uint16 *)stbi_errpuc("outofmem", "Out of memory");
1611     }
1612
1613     for (j = 0; j < (int)y; ++j) {
1614         stbi_uint16 *src = data + j * x * img_n;
1615         stbi_uint16 *dest = good + j * x * req_comp;
1616
1617 #define STBI_COMBO(a,b) ((a)*8+(b))
1618 #define STBI__CASE(a,b) case STBI_COMBO(a,b): for(i=x-1; i >= 0; --i, src += a, dest += b)
1619 // convert source image with img_n components to one with req_comp components;
1620 // avoid switch per pixel, so use switch per scanline and massive macros
1621 switch (STBI_COMBO(img_n, req_comp)) {
1622     STBI__CASE(1, 2) { dest[0] = src[0], dest[1] = 0xffff; } break;
1623     STBI__CASE(1, 3) { dest[0] = dest[1] = dest[2] = src[0]; } break;
1624     STBI__CASE(1, 4) { dest[0] = dest[1] = dest[2] = src[0], dest[3] = 0xffff; } break;
1625     STBI__CASE(2, 1) { dest[0] = src[0]; } break;
1626     STBI__CASE(2, 3) { dest[0] = dest[1] = dest[2] = src[0]; } break;
1627     STBI__CASE(2, 4) { dest[0] = dest[1] = src[0], dest[3] = src[1]; } break;
1628     STBI__CASE(3, 4) { dest[0] = src[0], dest[1] = src[1], dest[2] = src[2], dest[3] = 0xffff;
1629 } break;
1630     STBI__CASE(3, 1) { dest[0] = stbi_compute_y16(src[0], src[1], src[2]); } break;
1631     STBI__CASE(3, 2) { dest[0] = stbi_compute_y16(src[0], src[1], src[2]), dest[1] = 0xffff;
1632 } break;
1633     STBI__CASE(4, 1) { dest[0] = stbi_compute_y16(src[0], src[1], src[2]); } break;
1634     STBI__CASE(4, 2) { dest[0] = stbi_compute_y16(src[0], src[1], src[2]), dest[1] = src[3];
1635 } break;
1636     STBI__CASE(4, 3) { dest[0] = src[0], dest[1] = src[1], dest[2] = src[2]; } break;
1637     default: STBI_ASSERT(0);
1638 }
1639 #undef STBI__CASE
1640 #undef STBI_COMBO
1641
1642 STBI_FREE(data);
1643 return good;
1644 }
1645
1646 #ifndef STBI_NO_LINEAR
1647 static float *stbi_ldr_to_hdr(stbi_uc *data, int x, int y, int comp)
1648 {
1649     int i, k, n;
1650     float *output;
1651     if (!data) return NULL;
1652     output = (float *)stbi_malloc(sizeof(float), 0);
1653     if (output == NULL) { STBI_FREE(data); return stbi_errpuc("outofmem", "Out of memory"); }
1654     // compute number of non-alpha components
1655     if (comp & 1) n = comp; else n = comp - 1;
1656     for (i = 0; i < x*y; ++i) {
1657         for (k = 0; k < n; ++k) {
1658             output[i*comp + k] = (float)(pow(data[i*comp + k] / 255.0f, stbi__l2h_gamma) *
1659                 stbi__l2h_scale);
1660         }
1661         if (k < comp) output[i*comp + k] = data[i*comp + k] / 255.0f;
1662     }
1663     STBI_FREE(data);
1664     return output;
1665 }
1666 #endif
1667
1668 #ifndef STBI_NO_HDR
1669 #define stbi__float2int(x) ((int)(x))
1670 static stbi_uc *stbi_hdr_to_ldr(float *data, int x, int y, int comp)
1671 {
1672     int i, k, n;
1673     stbi_uc *output;
1674     if (!data) return NULL;
1675     output = (stbi_uc *)stbi_malloc(sizeof(stbi_uc), 0);
1676     if (output == NULL) { STBI_FREE(data); return stbi_errpuc("outofmem", "Out of memory"); }
1677     // compute number of non-alpha components
1678     if (comp & 1) n = comp; else n = comp - 1;
1679     for (i = 0; i < x*y; ++i) {
1680         for (k = 0; k < n; ++k) {
1681             float z = (float)pow(data[i*comp + k] * stbi__h2l_scale_i, stbi__h2l_gamma_i) * 255 + 0.5f;

```

```

1677         if (z < 0) z = 0;
1678         if (z > 255) z = 255;
1679         output[i*comp + k] = (stbi_uc)stbi__float2int(z);
1680     }
1681     if (k < comp) {
1682         float z = data[i*comp + k] * 255 + 0.5f;
1683         if (z < 0) z = 0;
1684         if (z > 255) z = 255;
1685         output[i*comp + k] = (stbi_uc)stbi__float2int(z);
1686     }
1687 }
1688 STBI_FREE(data);
1689 return output;
1690 }
1691 #endif
1692
1693 //
1694 // "baseline" JPEG/JFIF decoder
1695 //
1696 //
1697 //     simple implementation
1698 //     - doesn't support delayed output of y-dimension
1699 //     - simple interface (only one output format: 8-bit interleaved RGB)
1700 //     - doesn't try to recover corrupt jpegs
1701 //     - doesn't allow partial loading, loading multiple at once
1702 //     - still fast on x86 (copying globals into locals doesn't help x86)
1703 //     - allocates lots of intermediate memory (full size of all components)
1704 //     - non-interleaved case requires this anyway
1705 //     - allows good upsampling (see next)
1706 //
1707 // high-quality
1708 //     - upscaled channels are bilinearly interpolated, even across blocks
1709 //     - quality integer IDCT derived from IJG's 'slow'
1710 //
1711 // performance
1712 //     - fast huffman; reasonable integer IDCT
1713 //     - some SIMD kernels for common paths on targets with SSE2/NEON
1714 //     - uses a lot of intermediate memory, could cache poorly
1715
1716 #ifndef STBI_NO_JPEG
1717 // huffman decoding acceleration
1718 #define FAST_BITS 9 // larger handles more cases; smaller stomps less cache
1719
1720 typedef struct
1721 {
1722     stbi_uc fast[1 « FAST_BITS];
1723     // weirdly, repacking this into AoS is a 10% speed loss, instead of a win
1724     stbi_uint16 code[256];
1725     stbi_uc values[256];
1726     stbi_uc size[257];
1727     unsigned int maxcode[18];
1728     int delta[17]; // old 'firstsymbol' - old 'firstcode'
1729 } stbi__huffman;
1730
1731 typedef struct
1732 {
1733     stbi__context *s;
1734     stbi__huffman huff_dc[4];
1735     stbi__huffman huff_ac[4];
1736     stbi_uc dequant[4][64];
1737     stbi_int16 fast_ac[4][1 « FAST_BITS];
1738
1739     // sizes for components, interleaved MCUs
1740     int img_h_max, img_v_max;
1741     int img_mcu_x, img_mcu_y;
1742     int img_mcu_w, img_mcu_h;
1743
1744     // definition of jpeg image component
1745     struct
1746     {
1747         int id;
1748         int h, v;
1749         int tq;
1750         int hd, ha;
1751         int dc_pred;
1752
1753         int x, y, w2, h2;
1754         stbi_uc *data;
1755         void *raw_data, *raw_coeff;
1756         stbi_uc *linebuf;
1757         short *coeff; // progressive only
1758         int coeff_w, coeff_h; // number of 8x8 coefficient blocks
1759     } img_comp[4];
1760
1761     stbi_uint32 code_buffer; // jpeg entropy-coded buffer
1762     int code_bits; // number of valid bits
1763     unsigned char marker; // marker seen while filling entropy buffer
1764     int nomore; // flag if we saw a marker so must stop

```



```

1765     int             progressive;
1766     int             spec_start;
1767     int             spec_end;
1768     int             succ_high;
1769     int             succ_low;
1770     int             eob_run;
1771     int             rgb;
1772
1773     int scan_n, order[4];
1774     int restart_interval, todo;
1775
1776     // kernels
1777     void(*idct_block_kernel)(stbi_uc *out, int out_stride, short data[64]);
1778     void(*YCbCr_to_RGB_kernel)(stbi_uc *out, const stbi_uc *y, const stbi_uc *pcb, const stbi_uc *pcr,
1779     int count, int step);
1779     stbi_uc *(*resample_row_hv_2_kernel)(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int
1780     hs);
1780 } stbi__jpeg;
1781
1782 static int stbi__build_huffman(stbi__huffman *h, int *count)
1783 {
1784     int i, j, k = 0, code;
1785     // build size list for each symbol (from JPEG spec)
1786     for (i = 0; i < 16; ++i)
1787         for (j = 0; j < count[i]; ++j)
1788             h->size[k++] = (stbi_uc) (i + 1);
1789     h->size[k] = 0;
1790
1791     // compute actual symbols (from jpeg spec)
1792     code = 0;
1793     k = 0;
1794     for (j = 1; j <= 16; ++j) {
1795         // compute delta to add to code to compute symbol id
1796         h->delta[j] = k - code;
1797         if (h->size[k] == j) {
1798             while (h->size[k] == j)
1799                 h->code[k++] = (stbi_uint16) (code++);
1800             if (code - 1 >= (1 << j)) return stbi__err("bad code lengths", "Corrupt JPEG");
1801         }
1802         // compute largest code + 1 for this size, preshifted as needed later
1803         h->maxcode[j] = code << (16 - j);
1804         code <= 1;
1805     }
1806     h->maxcode[j] = 0xffffffff;
1807
1808     // build non-spec acceleration table; 255 is flag for not-accelerated
1809     memset(h->fast, 255, 1 << FAST_BITS);
1810     for (i = 0; i < k; ++i) {
1811         int s = h->size[i];
1812         if (s <= FAST_BITS) {
1813             int c = h->code[i] << (FAST_BITS - s);
1814             int m = 1 << (FAST_BITS - s);
1815             for (j = 0; j < m; ++j) {
1816                 h->fast[c + j] = (stbi_uc) i;
1817             }
1818         }
1819     }
1820     return 1;
1821 }
1822
1823 // build a table that decodes both magnitude and value of small ACs in
1824 // one go.
1825 static void stbi__build_fast_ac(stbi__int16 *fast_ac, stbi__huffman *h)
1826 {
1827     int i;
1828     for (i = 0; i < (1 << FAST_BITS); ++i) {
1829         stbi_uc fast = h->fast[i];
1830         fast_ac[i] = 0;
1831         if (fast < 255) {
1832             int rs = h->values[fast];
1833             int run = (rs >> 4) & 15;
1834             int magbits = rs & 15;
1835             int len = h->size[fast];
1836
1837             if (magbits && len + magbits <= FAST_BITS) {
1838                 // magnitude code followed by receive_extend code
1839                 int k = ((i < len) & ((1 << FAST_BITS) - 1)) >> (FAST_BITS - magbits);
1840                 int m = 1 << (magbits - 1);
1841                 if (k < m) k += (-1 << magbits) + 1;
1842                 // if the result is small enough, we can fit it in fast_ac table
1843                 if (k >= -128 && k <= 127)
1844                     fast_ac[i] = (stbi__int16) ((k << 8) + (run << 4) + (len + magbits));
1845             }
1846         }
1847     }
1848 }
1849

```

```

1850 static void stbi__grow_buffer_unsafe(stbi__jpeg *j)
1851 {
1852     do {
1853         int b = j->nomore ? 0 : stbi__get8(j->s);
1854         if (b == 0xff) {
1855             int c = stbi__get8(j->s);
1856             if (c != 0) {
1857                 j->marker = (unsigned char)c;
1858                 j->nomore = 1;
1859                 return;
1860             }
1861         }
1862         j->code_buffer |= b << (24 - j->code_bits);
1863         j->code_bits += 8;
1864     } while (j->code_bits <= 24);
1865 }
1866
1867 // (1 << n) - 1
1868 static stbi__uint32 stbi__bmask[17] = {
1869     0,1,3,7,15,31,63,127,255,511,1023,2047,4095,8191,16383,32767,65535 };
1870
1871 // decode a jpeg huffman value from the bitstream
1872 stbi_inline static int stbi__jpeg_huff_decode(stbi__jpeg *j, stbi__huffman *h)
1873 {
1874     unsigned int temp;
1875     int c, k;
1876
1877     if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
1878
1879     // look at the top FAST_BITS and determine what symbol ID it is,
1880     // if the code is <= FAST_BITS
1881     c = (j->code_buffer >> (32 - FAST_BITS)) & ((1 << FAST_BITS) - 1);
1882     k = h->fast[c];
1883     if (k < 255) {
1884         int s = h->size[k];
1885         if (s > j->code_bits)
1886             return -1;
1887         j->code_buffer <= s;
1888         j->code_bits -= s;
1889         return h->values[k];
1890     }
1891
1892     // naive test is to shift the code_buffer down so k bits are
1893     // valid, then test against maxcode. To speed this up, we've
1894     // preshifted maxcode left so that it has (16-k) 0s at the
1895     // end; in other words, regardless of the number of bits, it
1896     // wants to be compared against something shifted to have 16;
1897     // that way we don't need to shift inside the loop.
1898     temp = j->code_buffer >> 16;
1899     for (k = FAST_BITS + 1; ; ++k)
1900         if (temp < h->maxcode[k])
1901             break;
1902     if (k == 17) {
1903         // error! code not found
1904         j->code_bits -= 16;
1905         return -1;
1906     }
1907
1908     if (k > j->code_bits)
1909         return -1;
1910
1911     // convert the huffman code to the symbol id
1912     c = ((j->code_buffer >> (32 - k)) & stbi__bmask[k]) + h->delta[k];
1913     STBI_ASSERT((((j->code_buffer) >> (32 - h->size[c])) & stbi__bmask[h->size[c]]) == h->code[c]);
1914
1915     // convert the id to a symbol
1916     j->code_bits -= k;
1917     j->code_buffer <= k;
1918     return h->values[c];
1919 }
1920
1921 // bias[n] = (-1<n) + 1
1922 static int const stbi__jbias[16] = {
1923     0,-1,-3,-7,-15,-31,-63,-127,-255,-511,-1023,-2047,-4095,-8191,-16383,-32767 };
1924
1925 // combined JPEG 'receive' and JPEG 'extend', since baseline
1926 // always extends everything it receives.
1927 stbi_inline static int stbi__extend_receive(stbi__jpeg *j, int n)
1928 {
1929     unsigned int k;
1930     int sgn;
1931     if (j->code_bits < n) stbi__grow_buffer_unsafe(j);
1932
1933     sgn = (stbi__int32)j->code_buffer >> 31; // sign bit is always in MSB
1934     k = stbi_lrot(j->code_buffer, n);
1935     STBI_ASSERT(n >= 0 && n < (int)(sizeof(stbi__bmask) / sizeof(*stbi__bmask)));
1936     j->code_buffer = k & ~stbi__bmask[n];

```

```

1935     k &= stbi__bmask[n];
1936     j->code_bits -= n;
1937     return k + (stbi__jbias[n] & ~sgn);
1938 }
1939
1940 // get some unsigned bits
1941 stbi_inline static int stbi__jpeg_get_bits(stbi__jpeg *j, int n)
1942 {
1943     unsigned int k;
1944     if (j->code_bits < n) stbi__grow_buffer_unsafe(j);
1945     k = stbi_lrot(j->code_buffer, n);
1946     j->code_buffer = k & ~stbi__bmask[n];
1947     k &= stbi__bmask[n];
1948     j->code_bits -= n;
1949     return k;
1950 }
1951
1952 stbi_inline static int stbi__jpeg_get_bit(stbi__jpeg *j)
1953 {
1954     unsigned int k;
1955     if (j->code_bits < 1) stbi__grow_buffer_unsafe(j);
1956     k = j->code_buffer;
1957     j->code_buffer <<= 1;
1958     --j->code_bits;
1959     return k & 0x80000000;
1960 }
1961
1962 // given a value that's at position X in the zigzag stream,
1963 // where does it appear in the 8x8 matrix coded as row-major?
1964 static stbi_uc stbi__jpeg_dezigzag[64 + 15] =
1965 {
1966     0, 1, 8, 16, 9, 2, 3, 10,
1967     17, 24, 32, 25, 18, 11, 4, 5,
1968     12, 19, 26, 33, 40, 48, 41, 34,
1969     27, 20, 13, 6, 7, 14, 21, 28,
1970     35, 42, 49, 56, 57, 50, 43, 36,
1971     29, 22, 15, 23, 30, 37, 44, 51,
1972     58, 59, 52, 45, 38, 31, 39, 46,
1973     53, 60, 61, 54, 47, 55, 62, 63,
1974     // let corrupt input sample past end
1975     63, 63, 63, 63, 63, 63, 63, 63,
1976     63, 63, 63, 63, 63, 63, 63
1977 };
1978
1979 // decode one 64-entry block--
1980 static int stbi__jpeg_decode_block(stbi__jpeg *j, short data[64], stbi__huffman *hdc, stbi__huffman
    *hac, stbi__int16 *fac, int b, stbi_uc *dequant)
1981 {
1982     int diff, dc, k;
1983     int t;
1984
1985     if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
1986     t = stbi__jpeg_huff_decode(j, hdc);
1987     if (t < 0) return stbi__err("bad huffman code", "Corrupt JPEG");
1988
1989     // 0 all the ac values now so we can do it 32-bits at a time
1990     memset(data, 0, 64 * sizeof(data[0]));
1991
1992     diff = t ? stbi__extend_receive(j, t) : 0;
1993     dc = j->img_comp[b].dc_pred + diff;
1994     j->img_comp[b].dc_pred = dc;
1995     data[0] = (short)(dc * dequant[0]);
1996
1997     // decode AC components, see JPEG spec
1998     k = 1;
1999     do {
2000         unsigned int zig;
2001         int c, r, s;
2002         if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
2003         c = (j->code_buffer >> (32 - FAST_BITS)) & ((1 << FAST_BITS) - 1);
2004         r = fac[c];
2005         if (r) { // fast-AC path
2006             k += (r >> 4) & 15; // run
2007             s = r & 15; // combined length
2008             j->code_buffer <<= s;
2009             j->code_bits -= s;
2010             // decode into unzigzag'd location
2011             zig = stbi__jpeg_dezigzag[k++];
2012             data[zig] = (short)((r >> 8) * dequant[zig]);
2013         }
2014         else {
2015             int rs = stbi__jpeg_huff_decode(j, hac);
2016             if (rs < 0) return stbi__err("bad huffman code", "Corrupt JPEG");
2017             s = rs & 15;
2018             r = rs >> 4;
2019             if (s == 0) {
2020                 if (rs != 0xf0) break; // end block

```

```

2021         k += 16;
2022     }
2023     else {
2024         k += r;
2025         // decode into unzigzag'd location
2026         zig = stbi__jpeg_dezigzag[k++];
2027         data[zig] = (short)(stbi__extend_receive(j, s) * dequant[zig]);
2028     }
2029 }
2030 } while (k < 64);
2031 return 1;
2032 }
2033
2034 static int stbi__jpeg_decode_block_prog_dc(stbi__jpeg *j, short data[64], stbi__huffman *hdc, int b)
2035 {
2036     int diff, dc;
2037     int t;
2038     if (j->spec_end != 0) return stbi__err("can't merge dc and ac", "Corrupt JPEG");
2039
2040     if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
2041
2042     if (j->succ_high == 0) {
2043         // first scan for DC coefficient, must be first
2044         memset(data, 0, 64 * sizeof(data[0])); // 0 all the ac values now
2045         t = stbi__jpeg_huff_decode(j, hdc);
2046         diff = t ? stbi__extend_receive(j, t) : 0;
2047
2048         dc = j->img_comp[b].dc_pred + diff;
2049         j->img_comp[b].dc_pred = dc;
2050         data[0] = (short)(dc << j->succ_low);
2051     }
2052     else {
2053         // refinement scan for DC coefficient
2054         if (stbi__jpeg_get_bit(j))
2055             data[0] += (short)(1 << j->succ_low);
2056     }
2057     return 1;
2058 }
2059
2060 // @OPTIMIZE: store non-zigzagged during the decode passes,
2061 // and only de-zigzag when dequantizing
2062 static int stbi__jpeg_decode_block_prog_ac(stbi__jpeg *j, short data[64], stbi__huffman *hac,
2063     stbi__int16 *fac)
2064 {
2065     int k;
2066     if (j->spec_start == 0) return stbi__err("can't merge dc and ac", "Corrupt JPEG");
2067
2068     if (j->succ_high == 0) {
2069         int shift = j->succ_low;
2070
2071         if (j->eob_run) {
2072             --j->eob_run;
2073             return 1;
2074         }
2075
2076         k = j->spec_start;
2077         do {
2078             unsigned int zig;
2079             int c, r, s;
2080             if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
2081             c = (j->code_buffer >> (32 - FAST_BITS)) & ((1 << FAST_BITS) - 1);
2082             r = fac[c];
2083             if (r) { // fast-AC path
2084                 k += (r >> 4) & 15; // run
2085                 s = r & 15; // combined length
2086                 j->code_buffer <<= s;
2087                 j->code_bits -= s;
2088                 zig = stbi__jpeg_dezigzag[k++];
2089                 data[zig] = (short)((r >> 8) << shift);
2090             }
2091             else {
2092                 int rs = stbi__jpeg_huff_decode(j, hac);
2093                 if (rs < 0) return stbi__err("bad huffman code", "Corrupt JPEG");
2094                 s = rs & 15;
2095                 r = rs >> 4;
2096                 if (s == 0) {
2097                     if (r < 15) {
2098                         j->eob_run = (1 << r);
2099                         if (r)
2100                             j->eob_run += stbi__jpeg_get_bits(j, r);
2101                         --j->eob_run;
2102                         break;
2103                     }
2104                     k += 16;
2105                 }
2106                 else {
2107                     k += r;

```

```

2107         zig = stbi__jpeg_dezigzag[k++];
2108         data[zig] = (short)(stbi__extend_receive(j, s) << shift);
2109     }
2110 }
2111 } while (k <= j->spec_end);
2112 }
2113 else {
2114     // refinement scan for these AC coefficients
2115
2116     short bit = (short)(1 << j->succ_low);
2117
2118     if (j->eob_run) {
2119         --j->eob_run;
2120         for (k = j->spec_start; k <= j->spec_end; ++k) {
2121             short *p = &data[stbi__jpeg_dezigzag[k]];
2122             if (*p != 0)
2123                 if (stbi__jpeg_get_bit(j))
2124                     if ((*p & bit) == 0) {
2125                         if (*p > 0)
2126                             *p += bit;
2127                         else
2128                             *p -= bit;
2129                     }
2130         }
2131     }
2132     else {
2133         k = j->spec_start;
2134         do {
2135             int r, s;
2136             int rs = stbi__jpeg_huff_decode(j, hac); // @OPTIMIZE see if we can use the fast path
2137             here, advance-by-r is so slow, eh
2138             if (rs < 0) return stbi__err("bad huffman code", "Corrupt JPEG");
2139             s = rs & 15;
2140             r = rs >> 4;
2141             if (s == 0) {
2142                 if (r < 15) {
2143                     j->eob_run = (1 << r) - 1;
2144                     if (r)
2145                         j->eob_run += stbi__jpeg_get_bits(j, r);
2146                     r = 64; // force end of block
2147                 }
2148                 else {
2149                     // r=15 s=0 should write 16 0s, so we just do
2150                     // a run of 15 0s and then write s (which is 0),
2151                     // so we don't have to do anything special here
2152                 }
2153             }
2154             else {
2155                 if (s != 1) return stbi__err("bad huffman code", "Corrupt JPEG");
2156                 // sign bit
2157                 if (stbi__jpeg_get_bit(j))
2158                     s = bit;
2159                 else
2160                     s = -bit;
2161             }
2162             // advance by r
2163             while (k <= j->spec_end) {
2164                 short *p = &data[stbi__jpeg_dezigzag[k++]];
2165                 if (*p != 0) {
2166                     if (stbi__jpeg_get_bit(j))
2167                         if ((*p & bit) == 0) {
2168                             if (*p > 0)
2169                                 *p += bit;
2170                             else
2171                                 *p -= bit;
2172                         }
2173                     }
2174                 else {
2175                     if (r == 0) {
2176                         *p = (short)s;
2177                         break;
2178                     }
2179                     --r;
2180                 }
2181             }
2182         } while (k <= j->spec_end);
2183     }
2184 }
2185 return 1;
2186 }
2187
2188 // take a -128..127 value and stbi__clamp it and convert to 0..255
2189 stbi_inline static stbi_uc stbi__clamp(int x)
2190 {
2191     // trick to use a single test to catch both cases
2192     if ((unsigned int)x > 255) {

```

```

2193         if (x < 0) return 0;
2194         if (x > 255) return 255;
2195     }
2196     return (stbi_uc)x;
2197 }
2198
2199 #define stbi__f2f(x)  ((int) (((x) * 4096 + 0.5)))
2200 #define stbi__fsh(x)  ((x) << 12)
2201
2202 // derived from jidctint -- DCT_ISLOW
2203 #define STBI_IDCT_1D(s0,s1,s2,s3,s4,s5,s6,s7) \
2204     int t0,t1,t2,t3,p1,p2,p3,p4,p5,x0,x1,x2,x3; \
2205     p2 = s2; \
2206     p3 = s6; \
2207     p1 = (p2+p3) * stbi__f2f(0.5411961f); \
2208     t2 = p1 + p3*stbi__f2f(-1.847759065f); \
2209     t3 = p1 + p2*stbi__f2f( 0.765366865f); \
2210     p2 = s0; \
2211     p3 = s4; \
2212     t0 = stbi__fsh(p2+p3); \
2213     t1 = stbi__fsh(p2-p3); \
2214     x0 = t0+t3; \
2215     x3 = t0-t3; \
2216     x1 = t1+t2; \
2217     x2 = t1-t2; \
2218     t0 = s7; \
2219     t1 = s5; \
2220     t2 = s3; \
2221     t3 = s1; \
2222     p3 = t0+t2; \
2223     p4 = t1+t3; \
2224     p1 = t0+t3; \
2225     p2 = t1+t2; \
2226     p5 = (p3+p4)*stbi__f2f( 1.175875602f); \
2227     t0 = t0*stbi__f2f( 0.298631336f); \
2228     t1 = t1*stbi__f2f( 2.053119869f); \
2229     t2 = t2*stbi__f2f( 3.072711026f); \
2230     t3 = t3*stbi__f2f( 1.501321110f); \
2231     p1 = p5 + p1*stbi__f2f(-0.899976223f); \
2232     p2 = p5 + p2*stbi__f2f(-2.562915447f); \
2233     p3 = p3*stbi__f2f(-1.961570560f); \
2234     p4 = p4*stbi__f2f(-0.390180644f); \
2235     t3 += p1+p4; \
2236     t2 += p2+p3; \
2237     t1 += p2+p4; \
2238     t0 += p1+p3; \
2239
2240 static void stbi__idct_block(stbi_uc *out, int out_stride, short data[64])
2241 {
2242     int i, val[64], *v = val;
2243     stbi_uc *o;
2244     short *d = data;
2245
2246     // columns
2247     for (i = 0; i < 8; ++i, ++d, ++v) {
2248         // if all zeroes, shortcut -- this avoids dequantizing 0s and IDCTing
2249         if (d[8] == 0 && d[16] == 0 && d[24] == 0 && d[32] == 0
2250             && d[40] == 0 && d[48] == 0 && d[56] == 0) {
2251             // no shortcut
2252             // (1|2|3|4|5|6|7)==0      0      seconds
2253             // all separate              0      seconds
2254             // 1 && 2|3 && 4|5 && 6|7:  -0.047 seconds
2255             int dcterm = d[0] << 2;
2256             v[0] = v[8] = v[16] = v[24] = v[32] = v[40] = v[48] = v[56] = dcterm;
2257         }
2258         else {
2259             STBI_IDCT_1D(d[0], d[8], d[16], d[24], d[32], d[40], d[48], d[56])
2260             // constants scaled things up by 1<12; let's bring them back
2261             // down, but keep 2 extra bits of precision
2262             x0 += 512; x1 += 512; x2 += 512; x3 += 512;
2263             v[0] = (x0 + t3) >> 10;
2264             v[56] = (x0 - t3) >> 10;
2265             v[8] = (x1 + t2) >> 10;
2266             v[48] = (x1 - t2) >> 10;
2267             v[16] = (x2 + t1) >> 10;
2268             v[40] = (x2 - t1) >> 10;
2269             v[24] = (x3 + t0) >> 10;
2270             v[32] = (x3 - t0) >> 10;
2271         }
2272     }
2273
2274     for (i = 0, v = val, o = out; i < 8; ++i, v += 8, o += out_stride) {
2275         // no fast case since the first 1D IDCT spread components out
2276         STBI_IDCT_1D(v[0], v[1], v[2], v[3], v[4], v[5], v[6], v[7])
2277         // constants scaled things up by 1<12, plus we had 1<2 from first
2278         // loop, plus horizontal and vertical each scale by sqrt(8) so together
2279         // we've got an extra 1<3, so 1<17 total we need to remove.

```

```

2280         // so we want to round that, which means adding 0.5 * 1<17,
2281         // aka 65536. Also, we'll end up with -128 to 127 that we want
2282         // to encode as 0..255 by adding 128, so we'll add that before the shift
2283         x0 += 65536 + (128 < 17);
2284         x1 += 65536 + (128 < 17);
2285         x2 += 65536 + (128 < 17);
2286         x3 += 65536 + (128 < 17);
2287         // tried computing the shifts into temps, or'ing the temps to see
2288         // if any were out of range, but that was slower
2289         o[0] = stbi__clamp((x0 + t3) >> 17);
2290         o[7] = stbi__clamp((x0 - t3) >> 17);
2291         o[1] = stbi__clamp((x1 + t2) >> 17);
2292         o[6] = stbi__clamp((x1 - t2) >> 17);
2293         o[2] = stbi__clamp((x2 + t1) >> 17);
2294         o[5] = stbi__clamp((x2 - t1) >> 17);
2295         o[3] = stbi__clamp((x3 + t0) >> 17);
2296         o[4] = stbi__clamp((x3 - t0) >> 17);
2297     }
2298 }
2299
2300 #ifdef STBI_SSE2
2301 // sse2 integer IDCT. not the fastest possible implementation but it
2302 // produces bit-identical results to the generic C version so it's
2303 // fully "transparent".
2304 static void stbi__idct_simd(stbi_uc *out, int out_stride, short data[64])
2305 {
2306     // This is constructed to match our regular (generic) integer IDCT exactly.
2307     __m128i row0, row1, row2, row3, row4, row5, row6, row7;
2308     __m128i tmp;
2309
2310     // dot product constant: even elems=x, odd elems=y
2311     #define dct_const(x,y)  _mm_setr_epi16((x),(y),(x),(y),(x),(y),(x),(y))
2312
2313     // out(0) = c0[even]*x + c0[odd]*y    (c0, x, y 16-bit, out 32-bit)
2314     // out(1) = c1[even]*x + c1[odd]*y
2315     #define dct_rot(out0,out1, x,y,c0,c1) \
2316         __m128i c0##lo = _mm_unpacklo_epi16((x),(y)); \
2317         __m128i c0##hi = _mm_unpackhi_epi16((x),(y)); \
2318         __m128i out0##l = _mm_madd_epi16(c0##lo, c0); \
2319         __m128i out0##h = _mm_madd_epi16(c0##hi, c0); \
2320         __m128i out1##l = _mm_madd_epi16(c0##lo, c1); \
2321         __m128i out1##h = _mm_madd_epi16(c0##hi, c1)
2322
2323     // out = in << 12 (in 16-bit, out 32-bit)
2324     #define dct_widen(out, in) \
2325         __m128i out##_l = _mm_srai_epi32(_mm_unpacklo_epi16(_mm_setzero_si128(), (in)), 4); \
2326         __m128i out##_h = _mm_srai_epi32(_mm_unpackhi_epi16(_mm_setzero_si128(), (in)), 4)
2327
2328     // wide add
2329     #define dct_wadd(out, a, b) \
2330         __m128i out##_l = _mm_add_epi32(a##_l, b##_l); \
2331         __m128i out##_h = _mm_add_epi32(a##_h, b##_h)
2332
2333     // wide sub
2334     #define dct_wsub(out, a, b) \
2335         __m128i out##_l = _mm_sub_epi32(a##_l, b##_l); \
2336         __m128i out##_h = _mm_sub_epi32(a##_h, b##_h)
2337
2338     // butterfly a/b, add bias, then shift by "s" and pack
2339     #define dct_bfly32o(out0, out1, a,b,bias,s) \
2340         { \
2341             __m128i abias0_l = _mm_add_epi32(a##_l, bias); \
2342             __m128i abias0_h = _mm_add_epi32(a##_h, bias); \
2343             dct_wadd(sum, abias0, b); \
2344             dct_wsub(dif, abias0, b); \
2345             out0 = _mm_packs_epi32(_mm_srai_epi32(sum_l, s), _mm_srai_epi32(sum_h, s)); \
2346             out1 = _mm_packs_epi32(_mm_srai_epi32(dif_l, s), _mm_srai_epi32(dif_h, s)); \
2347         }
2348
2349     // 8-bit interleave step (for transposes)
2350     #define dct_interleave8(a, b) \
2351         tmp = a; \
2352         a = _mm_unpacklo_epi8(a, b); \
2353         b = _mm_unpackhi_epi8(tmp, b)
2354
2355     // 16-bit interleave step (for transposes)
2356     #define dct_interleave16(a, b) \
2357         tmp = a; \
2358         a = _mm_unpacklo_epi16(a, b); \
2359         b = _mm_unpackhi_epi16(tmp, b)
2360
2361     #define dct_pass(bias,shift) \
2362         { \
2363             /* even part */ \
2364             dct_rot(t2e,t3e, row2,row6, rot0_0,rot0_1); \
2365             __m128i sum04 = _mm_add_epi16(row0, row4); \
2366             __m128i dif04 = _mm_sub_epi16(row0, row4); \

```

```

2367     dct_widen(t0e, sum04); \
2368     dct_widen(t1e, dif04); \
2369     dct_wadd(x0, t0e, t3e); \
2370     dct_wsub(x3, t0e, t3e); \
2371     dct_wadd(x1, t1e, t2e); \
2372     dct_wsub(x2, t1e, t2e); \
2373     /* odd part */ \
2374     dct_rot(y0o,y2o, row7,row3, rot2_0,rot2_1); \
2375     dct_rot(y1o,y3o, row5,row1, rot3_0,rot3_1); \
2376     __m128i sum17 = _mm_add_epi16(row1, row7); \
2377     __m128i sum35 = _mm_add_epi16(row3, row5); \
2378     dct_rot(y4o,y5o, sum17,sum35, rot1_0,rot1_1); \
2379     dct_wadd(x4, y0o, y4o); \
2380     dct_wadd(x5, y1o, y5o); \
2381     dct_wadd(x6, y2o, y5o); \
2382     dct_wadd(x7, y3o, y4o); \
2383     dct_bfly32o(row0,row7, x0,x7,bias,shift); \
2384     dct_bfly32o(row1,row6, x1,x6,bias,shift); \
2385     dct_bfly32o(row2,row5, x2,x5,bias,shift); \
2386     dct_bfly32o(row3,row4, x3,x4,bias,shift); \
2387 }
2388
2389     __m128i rot0_0 = dct_const(stbi__f2f(0.5411961f), stbi__f2f(0.5411961f) +
stbi__f2f(-1.847759065f));
2390     __m128i rot0_1 = dct_const(stbi__f2f(0.5411961f) + stbi__f2f(0.765366865f), stbi__f2f(0.5411961f));
2391     __m128i rot1_0 = dct_const(stbi__f2f(1.175875602f) + stbi__f2f(-0.899976223f),
stbi__f2f(1.175875602f));
2392     __m128i rot1_1 = dct_const(stbi__f2f(1.175875602f), stbi__f2f(1.175875602f) +
stbi__f2f(-2.562915447f));
2393     __m128i rot2_0 = dct_const(stbi__f2f(-1.961570560f) + stbi__f2f(0.298631336f),
stbi__f2f(-1.961570560f));
2394     __m128i rot2_1 = dct_const(stbi__f2f(-1.961570560f), stbi__f2f(-1.961570560f) +
stbi__f2f(3.072711026f));
2395     __m128i rot3_0 = dct_const(stbi__f2f(-0.390180644f) + stbi__f2f(2.053119869f),
stbi__f2f(-0.390180644f));
2396     __m128i rot3_1 = dct_const(stbi__f2f(-0.390180644f), stbi__f2f(-0.390180644f) +
stbi__f2f(1.501321110f));
2397
2398     // rounding biases in column/row passes, see stbi__idct_block for explanation.
2399     __m128i bias_0 = _mm_set1_epi32(512);
2400     __m128i bias_1 = _mm_set1_epi32(65536 + (128 < 17));
2401
2402     // load
2403     row0 = _mm_load_si128((const __m128i *) (data + 0 * 8));
2404     row1 = _mm_load_si128((const __m128i *) (data + 1 * 8));
2405     row2 = _mm_load_si128((const __m128i *) (data + 2 * 8));
2406     row3 = _mm_load_si128((const __m128i *) (data + 3 * 8));
2407     row4 = _mm_load_si128((const __m128i *) (data + 4 * 8));
2408     row5 = _mm_load_si128((const __m128i *) (data + 5 * 8));
2409     row6 = _mm_load_si128((const __m128i *) (data + 6 * 8));
2410     row7 = _mm_load_si128((const __m128i *) (data + 7 * 8));
2411
2412     // column pass
2413     dct_pass(bias_0, 10);
2414
2415     {
2416         // 16bit 8x8 transpose pass 1
2417         dct_interleave16(row0, row4);
2418         dct_interleave16(row1, row5);
2419         dct_interleave16(row2, row6);
2420         dct_interleave16(row3, row7);
2421
2422         // transpose pass 2
2423         dct_interleave16(row0, row2);
2424         dct_interleave16(row1, row3);
2425         dct_interleave16(row4, row6);
2426         dct_interleave16(row5, row7);
2427
2428         // transpose pass 3
2429         dct_interleave16(row0, row1);
2430         dct_interleave16(row2, row3);
2431         dct_interleave16(row4, row5);
2432         dct_interleave16(row6, row7);
2433     }
2434
2435     // row pass
2436     dct_pass(bias_1, 17);
2437
2438     {
2439         // pack
2440         __m128i p0 = _mm_packus_epi16(row0, row1); // a0a1a2a3...a7b0b1b2b3...b7
2441         __m128i p1 = _mm_packus_epi16(row2, row3);
2442         __m128i p2 = _mm_packus_epi16(row4, row5);
2443         __m128i p3 = _mm_packus_epi16(row6, row7);
2444
2445         // 8bit 8x8 transpose pass 1
2446         dct_interleave8(p0, p2); // a0e0a1e1...

```



```

2447     dct_interleave8(p1, p3); // c0g0c1g1...
2448
2449     // transpose pass 2
2450     dct_interleave8(p0, p1); // a0c0e0g0...
2451     dct_interleave8(p2, p3); // b0d0f0h0...
2452
2453     // transpose pass 3
2454     dct_interleave8(p0, p2); // a0b0c0d0...
2455     dct_interleave8(p1, p3); // a4b4c4d4...
2456
2457     // store
2458     _mm_storel_epi64((__m128i *) out, p0); out += out_stride;
2459     _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p0, 0x4e)); out += out_stride;
2460     _mm_storel_epi64((__m128i *) out, p2); out += out_stride;
2461     _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p2, 0x4e)); out += out_stride;
2462     _mm_storel_epi64((__m128i *) out, p1); out += out_stride;
2463     _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p1, 0x4e)); out += out_stride;
2464     _mm_storel_epi64((__m128i *) out, p3); out += out_stride;
2465     _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p3, 0x4e));
2466 }
2467
2468 #undef dct_const
2469 #undef dct_rot
2470 #undef dct_widen
2471 #undef dct_wadd
2472 #undef dct_wsub
2473 #undef dct_bfly32o
2474 #undef dct_interleave8
2475 #undef dct_interleave16
2476 #undef dct_pass
2477 }
2478
2479 #endif // STBI_SSE2
2480
2481 #ifdef STBI_NEON
2482
2483 // NEON integer IDCT. should produce bit-identical
2484 // results to the generic C version.
2485 static void stbi__idct_simd(stbi_uc *out, int out_stride, short data[64])
2486 {
2487     int16x8_t row0, row1, row2, row3, row4, row5, row6, row7;
2488
2489     int16x4_t rot0_0 = vdup_n_s16(stbi__f2f(0.5411961f));
2490     int16x4_t rot0_1 = vdup_n_s16(stbi__f2f(-1.847759065f));
2491     int16x4_t rot0_2 = vdup_n_s16(stbi__f2f(0.765366865f));
2492     int16x4_t rot1_0 = vdup_n_s16(stbi__f2f(1.175875602f));
2493     int16x4_t rot1_1 = vdup_n_s16(stbi__f2f(-0.899976223f));
2494     int16x4_t rot1_2 = vdup_n_s16(stbi__f2f(-2.562915447f));
2495     int16x4_t rot2_0 = vdup_n_s16(stbi__f2f(-1.961570560f));
2496     int16x4_t rot2_1 = vdup_n_s16(stbi__f2f(-0.390180644f));
2497     int16x4_t rot3_0 = vdup_n_s16(stbi__f2f(0.298631336f));
2498     int16x4_t rot3_1 = vdup_n_s16(stbi__f2f(2.053119869f));
2499     int16x4_t rot3_2 = vdup_n_s16(stbi__f2f(3.072711026f));
2500     int16x4_t rot3_3 = vdup_n_s16(stbi__f2f(1.501321110f));
2501
2502 #define dct_long_mul(out, inq, coeff) \
2503     int32x4_t out##_l = vmull_s16(vget_low_s16(inq), coeff); \
2504     int32x4_t out##_h = vmull_s16(vget_high_s16(inq), coeff)
2505
2506 #define dct_long_mac(out, acc, inq, coeff) \
2507     int32x4_t out##_l = vmlal_s16(acc##_l, vget_low_s16(inq), coeff); \
2508     int32x4_t out##_h = vmlal_s16(acc##_h, vget_high_s16(inq), coeff)
2509
2510 #define dct_widen(out, inq) \
2511     int32x4_t out##_l = vshll_n_s16(vget_low_s16(inq), 12); \
2512     int32x4_t out##_h = vshll_n_s16(vget_high_s16(inq), 12)
2513
2514 // wide add
2515 #define dct_wadd(out, a, b) \
2516     int32x4_t out##_l = vaddq_s32(a##_l, b##_l); \
2517     int32x4_t out##_h = vaddq_s32(a##_h, b##_h)
2518
2519 // wide sub
2520 #define dct_wsub(out, a, b) \
2521     int32x4_t out##_l = vsubq_s32(a##_l, b##_l); \
2522     int32x4_t out##_h = vsubq_s32(a##_h, b##_h)
2523
2524 // butterfly a/b, then shift using "shifto" by "s" and pack
2525 #define dct_bfly32o(out0,out1, a,b,shifto,s) \
2526 { \
2527     dct_wadd(sum, a, b); \
2528     dct_wsub(dif, a, b); \
2529     out0 = vcombine_s16(shifto(sum_l, s), shifto(sum_h, s)); \
2530     out1 = vcombine_s16(shifto(dif_l, s), shifto(dif_h, s)); \
2531 }
2532
2533 #define dct_pass(shifto, shift) \

```

```

2534 { \
2535     /* even part */ \
2536     int16x8_t sum26 = vaddq_s16(row2, row6); \
2537     dct_long_mul(p1e, sum26, rot0_0); \
2538     dct_long_mac(t2e, p1e, row6, rot0_1); \
2539     dct_long_mac(t3e, p1e, row2, rot0_2); \
2540     int16x8_t sum04 = vaddq_s16(row0, row4); \
2541     int16x8_t dif04 = vsubq_s16(row0, row4); \
2542     dct_widen(t0e, sum04); \
2543     dct_widen(t1e, dif04); \
2544     dct_wadd(x0, t0e, t3e); \
2545     dct_wsub(x3, t0e, t3e); \
2546     dct_wadd(x1, t1e, t2e); \
2547     dct_wsub(x2, t1e, t2e); \
2548     /* odd part */ \
2549     int16x8_t sum15 = vaddq_s16(row1, row5); \
2550     int16x8_t sum17 = vaddq_s16(row1, row7); \
2551     int16x8_t sum35 = vaddq_s16(row3, row5); \
2552     int16x8_t sum37 = vaddq_s16(row3, row7); \
2553     int16x8_t sumodd = vaddq_s16(sum17, sum35); \
2554     dct_long_mul(p5o, sumodd, rot1_0); \
2555     dct_long_mac(p1o, p5o, sum17, rot1_1); \
2556     dct_long_mac(p2o, p5o, sum35, rot1_2); \
2557     dct_long_mul(p3o, sum37, rot2_0); \
2558     dct_long_mul(p4o, sum15, rot2_1); \
2559     dct_wadd(sump13o, p1o, p3o); \
2560     dct_wadd(sump24o, p2o, p4o); \
2561     dct_wadd(sump23o, p2o, p3o); \
2562     dct_wadd(sump14o, p1o, p4o); \
2563     dct_long_mac(x4, sump13o, row7, rot3_0); \
2564     dct_long_mac(x5, sump24o, row5, rot3_1); \
2565     dct_long_mac(x6, sump23o, row3, rot3_2); \
2566     dct_long_mac(x7, sump14o, row1, rot3_3); \
2567     dct_bfly32o(row0, row7, x0, x7, shifto, shift); \
2568     dct_bfly32o(row1, row6, x1, x6, shifto, shift); \
2569     dct_bfly32o(row2, row5, x2, x5, shifto, shift); \
2570     dct_bfly32o(row3, row4, x3, x4, shifto, shift); \
2571 }
2572
2573 // load
2574 row0 = vld1q_s16(data + 0 * 8);
2575 row1 = vld1q_s16(data + 1 * 8);
2576 row2 = vld1q_s16(data + 2 * 8);
2577 row3 = vld1q_s16(data + 3 * 8);
2578 row4 = vld1q_s16(data + 4 * 8);
2579 row5 = vld1q_s16(data + 5 * 8);
2580 row6 = vld1q_s16(data + 6 * 8);
2581 row7 = vld1q_s16(data + 7 * 8);
2582
2583 // add DC bias
2584 row0 = vaddq_s16(row0, vsetq_lane_s16(1024, vdupq_n_s16(0), 0));
2585
2586 // column pass
2587 dct_pass(vrshrn_n_s32, 10);
2588
2589 // 16bit 8x8 transpose
2590 {
2591     // these three map to a single VTRN.16, VTRN.32, and VSWP, respectively.
2592     // whether compilers actually get this is another story, sadly.
2593 #define dct_trn16(x, y) { int16x8_t t = vtrnq_s16(x, y); x = t.val[0]; y = t.val[1]; }
2594 #define dct_trn32(x, y) { int32x4_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]); }
2595 #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)); }
2596
2597 // pass 1
2598 dct_trn16(row0, row1); // a0b0a2b2a4b4a6b6
2599 dct_trn16(row2, row3);
2600 dct_trn16(row4, row5);
2601 dct_trn16(row6, row7);
2602
2603 // pass 2
2604 dct_trn32(row0, row2); // a0b0c0d0a4b4c4d4
2605 dct_trn32(row1, row3);
2606 dct_trn32(row4, row6);
2607 dct_trn32(row5, row7);
2608
2609 // pass 3
2610 dct_trn64(row0, row4); // a0b0c0d0e0f0g0h0
2611 dct_trn64(row1, row5);
2612 dct_trn64(row2, row6);
2613 dct_trn64(row3, row7);
2614
2615 #undef dct_trn16
2616 #undef dct_trn32
2617 #undef dct_trn64

```

```

2618     }
2619
2620     // row pass
2621     // vrshrn_n_s32 only supports shifts up to 16, we need
2622     // 17. so do a non-rounding shift of 16 first then follow
2623     // up with a rounding shift by 1.
2624     dct_pass(vrshrn_n_s32, 16);
2625
2626     {
2627         // pack and round
2628         uint8x8_t p0 = vqrshrun_n_s16(row0, 1);
2629         uint8x8_t p1 = vqrshrun_n_s16(row1, 1);
2630         uint8x8_t p2 = vqrshrun_n_s16(row2, 1);
2631         uint8x8_t p3 = vqrshrun_n_s16(row3, 1);
2632         uint8x8_t p4 = vqrshrun_n_s16(row4, 1);
2633         uint8x8_t p5 = vqrshrun_n_s16(row5, 1);
2634         uint8x8_t p6 = vqrshrun_n_s16(row6, 1);
2635         uint8x8_t p7 = vqrshrun_n_s16(row7, 1);
2636
2637         // again, these can translate into one instruction, but often don't.
2638         #define dct_trn8_8(x, y) { uint8x8x2_t t = vtrn_u8(x, y); x = t.val[0]; y = t.val[1]; }
2639         #define dct_trn8_16(x, y) { uint16x4x2_t t = vtrn_u16(vreinterpret_u16_u8(x), reinterpret_u16_u8(y));
2640         #define dct_trn8_32(x, y) { uint32x2x2_t t = vtrn_u32(vreinterpret_u32_u8(x), reinterpret_u32_u8(y));
2641         x = reinterpret_u8_u32(t.val[0]); y = reinterpret_u8_u32(t.val[1]); }
2642
2643         // sadly can't use interleaved stores here since we only write
2644         // 8 bytes to each scan line!
2645
2646         // 8x8 8-bit transpose pass 1
2647         dct_trn8_8(p0, p1);
2648         dct_trn8_8(p2, p3);
2649         dct_trn8_8(p4, p5);
2650         dct_trn8_8(p6, p7);
2651
2652         // pass 2
2653         dct_trn8_16(p0, p2);
2654         dct_trn8_16(p1, p3);
2655         dct_trn8_16(p4, p6);
2656         dct_trn8_16(p5, p7);
2657
2658         // pass 3
2659         dct_trn8_32(p0, p4);
2660         dct_trn8_32(p1, p5);
2661         dct_trn8_32(p2, p6);
2662         dct_trn8_32(p3, p7);
2663
2664         // store
2665         vst1_u8(out, p0); out += out_stride;
2666         vst1_u8(out, p1); out += out_stride;
2667         vst1_u8(out, p2); out += out_stride;
2668         vst1_u8(out, p3); out += out_stride;
2669         vst1_u8(out, p4); out += out_stride;
2670         vst1_u8(out, p5); out += out_stride;
2671         vst1_u8(out, p6); out += out_stride;
2672         vst1_u8(out, p7);
2673
2674         #undef dct_trn8_8
2675         #undef dct_trn8_16
2676         #undef dct_trn8_32
2677     }
2678
2679     #undef dct_long_mul
2680     #undef dct_long_mac
2681     #undef dct_widen
2682     #undef dct_wadd
2683     #undef dct_wsub
2684     #undef dct_bfly32o
2685     #undef dct_pass
2686 }
2687
2688 #endif // STBI_NEON
2689
2690 #define STBI__MARKER_none 0xff
2691 // if there's a pending marker from the entropy stream, return that
2692 // otherwise, fetch from the stream and get a marker. if there's no
2693 // marker, return 0xff, which is never a valid marker value
2694 static stbi_uc stbi__get_marker(stbi__jpeg *j)
2695 {
2696     stbi_uc x;
2697     if (j->marker != STBI__MARKER_none) { x = j->marker; j->marker = STBI__MARKER_none; return x; }
2698     x = stbi__get8(j->s);
2699     if (x != 0xff) return STBI__MARKER_none;
2700     while (x == 0xff)
2701         x = stbi__get8(j->s);
2702     return x;
2703 }

```

```

2703
2704 // in each scan, we'll have scan_n components, and the order
2705 // of the components is specified by order[]
2706 #define STBI__RESTART(x) ((x) >= 0xd0 && (x) <= 0xd7)
2707
2708 // after a restart interval, stbi__jpeg_reset the entropy decoder and
2709 // the dc prediction
2710 static void stbi__jpeg_reset(stbi__jpeg *j)
2711 {
2712     j->code_bits = 0;
2713     j->code_buffer = 0;
2714     j->nomore = 0;
2715     j->img_comp[0].dc_pred = j->img_comp[1].dc_pred = j->img_comp[2].dc_pred = 0;
2716     j->marker = STBI__MARKER_none;
2717     j->todo = j->restart_interval ? j->restart_interval : 0xffffffff;
2718     j->eob_run = 0;
2719     // no more than 1<31 MCUs if no restart_interval? that's plenty safe,
2720     // since we don't even allow 1<30 pixels
2721 }
2722
2723 static int stbi__parse_entropy_coded_data(stbi__jpeg *z)
2724 {
2725     stbi__jpeg_reset(z);
2726     if (!z->progressive) {
2727         if (z->scan_n == 1) {
2728             int i, j;
2729             STBI_SIMD_ALIGN(short, data[64]);
2730             int n = z->order[0];
2731             // non-interleaved data, we just need to process one block at a time,
2732             // in trivial scanline order
2733             // number of blocks to do just depends on how many actual "pixels" this
2734             // component has, independent of interleaved MCU blocking and such
2735             int w = (z->img_comp[n].x + 7) >> 3;
2736             int h = (z->img_comp[n].y + 7) >> 3;
2737             for (j = 0; j < h; ++j) {
2738                 for (i = 0; i < w; ++i) {
2739                     int ha = z->img_comp[n].ha;
2740                     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;
2741                     z->idct_block_kernel(z->img_comp[n].data + z->img_comp[n].w2*j * 8 + i * 8,
z->img_comp[n].w2, data);
2742                     // every data block is an MCU, so countdown the restart interval
2743                     if (--z->todo <= 0) {
2744                         if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
2745                         // if it's NOT a restart, then just bail, so we get corrupt data
2746                         // rather than no data
2747                         if (!STBI__RESTART(z->marker)) return 1;
2748                         stbi__jpeg_reset(z);
2749                     }
2750                 }
2751             }
2752             return 1;
2753         }
2754         else { // interleaved
2755             int i, j, k, x, y;
2756             STBI_SIMD_ALIGN(short, data[64]);
2757             for (j = 0; j < z->img_mcu_y; ++j) {
2758                 for (i = 0; i < z->img_mcu_x; ++i) {
2759                     // scan an interleaved mcu... process scan_n components in order
2760                     for (k = 0; k < z->scan_n; ++k) {
2761                         int n = z->order[k];
2762                         // scan out an mcu's worth of this component; that's just determined
2763                         // by the basic H and V specified for the component
2764                         for (y = 0; y < z->img_comp[n].v; ++y) {
2765                             for (x = 0; x < z->img_comp[n].h; ++x) {
2766                                 int x2 = (i*z->img_comp[n].h + x) * 8;
2767                                 int y2 = (j*z->img_comp[n].v + y) * 8;
2768                                 int ha = z->img_comp[n].ha;
2769                                 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;
2770                                 z->idct_block_kernel(z->img_comp[n].data + z->img_comp[n].w2*y2 + x2,
z->img_comp[n].w2, data);
2771                             }
2772                         }
2773                     }
2774                     // after all interleaved components, that's an interleaved MCU,
2775                     // so now count down the restart interval
2776                     if (--z->todo <= 0) {
2777                         if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
2778                         if (!STBI__RESTART(z->marker)) return 1;
2779                         stbi__jpeg_reset(z);
2780                     }
2781                 }
2782             }
2783             return 1;
2784         }
2785     }

```

```

2786     else {
2787         if (z->scan_n == 1) {
2788             int i, j;
2789             int n = z->order[0];
2790             // non-interleaved data, we just need to process one block at a time,
2791             // in trivial scanline order
2792             // number of blocks to do just depends on how many actual "pixels" this
2793             // component has, independent of interleaved MCU blocking and such
2794             int w = (z->img_comp[n].x + 7) >> 3;
2795             int h = (z->img_comp[n].y + 7) >> 3;
2796             for (j = 0; j < h; ++j) {
2797                 for (i = 0; i < w; ++i) {
2798                     short *data = z->img_comp[n].coeff + 64 * (i + j * z->img_comp[n].coeff_w);
2799                     if (z->spec_start == 0) {
2800                         if (!stbi__jpeg_decode_block_prog_dc(z, data, &z->huff_dc[z->img_comp[n].hd],
n))
2801                             return 0;
2802                     }
2803                     else {
2804                         int ha = z->img_comp[n].ha;
2805                         if (!stbi__jpeg_decode_block_prog_ac(z, data, &z->huff_ac[ha], z->fast_ac[ha]))
2806                             return 0;
2807                     }
2808                     // every data block is an MCU, so countdown the restart interval
2809                     if (--z->todo <= 0) {
2810                         if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
2811                         if (!STBI__RESTART(z->marker)) return 1;
2812                         stbi__jpeg_reset(z);
2813                     }
2814                 }
2815             }
2816             return 1;
2817         }
2818         else { // interleaved
2819             int i, j, k, x, y;
2820             for (j = 0; j < z->img_mcu_y; ++j) {
2821                 for (i = 0; i < z->img_mcu_x; ++i) {
2822                     // scan an interleaved mcu... process scan_n components in order
2823                     for (k = 0; k < z->scan_n; ++k) {
2824                         int n = z->order[k];
2825                         // scan out an mcu's worth of this component; that's just determined
2826                         // by the basic H and V specified for the component
2827                         for (y = 0; y < z->img_comp[n].v; ++y) {
2828                             for (x = 0; x < z->img_comp[n].h; ++x) {
2829                                 int x2 = (i+z->img_comp[n].h + x);
2830                                 int y2 = (j+z->img_comp[n].v + y);
2831                                 short *data = z->img_comp[n].coeff + 64 * (x2 + y2 *
z->img_comp[n].coeff_w);
2832                                 if (!stbi__jpeg_decode_block_prog_dc(z, data,
&z->huff_dc[z->img_comp[n].hd], n))
2833                                     return 0;
2834                             }
2835                         }
2836                     }
2837                     // after all interleaved components, that's an interleaved MCU,
2838                     // so now count down the restart interval
2839                     if (--z->todo <= 0) {
2840                         if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
2841                         if (!STBI__RESTART(z->marker)) return 1;
2842                         stbi__jpeg_reset(z);
2843                     }
2844                 }
2845             }
2846             return 1;
2847         }
2848     }
2849 }
2850
2851 static void stbi__jpeg_dequantize(short *data, stbi_uc *dequant)
2852 {
2853     int i;
2854     for (i = 0; i < 64; ++i)
2855         data[i] *= dequant[i];
2856 }
2857
2858 static void stbi__jpeg_finish(stbi__jpeg *z)
2859 {
2860     if (z->progressive) {
2861         // dequantize and idct the data
2862         int i, j, n;
2863         for (n = 0; n < z->s->img_n; ++n) {
2864             int w = (z->img_comp[n].x + 7) >> 3;
2865             int h = (z->img_comp[n].y + 7) >> 3;
2866             for (j = 0; j < h; ++j) {
2867                 for (i = 0; i < w; ++i) {
2868                     short *data = z->img_comp[n].coeff + 64 * (i + j * z->img_comp[n].coeff_w);
2869                     stbi__jpeg_dequantize(data, z->dequant[z->img_comp[n].tq]);

```

```

2870         z->idct_block_kernel(z->img_comp[n].data + z->img_comp[n].w2*j * 8 + i * 8,
2871         z->img_comp[n].w2, data);
2872     }
2873 }
2874 }
2875 }
2876
2877 static int stbi__process_marker(stbi__jpeg *z, int m)
2878 {
2879     int L;
2880     switch (m) {
2881     case STBI__MARKER_none: // no marker found
2882         return stbi__err("expected marker", "Corrupt JPEG");
2883
2884     case 0xDD: // DRI - specify restart interval
2885         if (stbi__get16be(z->s) != 4) return stbi__err("bad DRI len", "Corrupt JPEG");
2886         z->restart_interval = stbi__get16be(z->s);
2887         return 1;
2888
2889     case 0xDB: // DQT - define quantization table
2890         L = stbi__get16be(z->s) - 2;
2891         while (L > 0) {
2892             int q = stbi__get8(z->s);
2893             int p = q >> 4;
2894             int t = q & 15, i;
2895             if (p != 0) return stbi__err("bad DQT type", "Corrupt JPEG");
2896             if (t > 3) return stbi__err("bad DQT table", "Corrupt JPEG");
2897             for (i = 0; i < 64; ++i)
2898                 z->dequant[t][stbi__jpeg_dezigzag[i]] = stbi__get8(z->s);
2899             L -= 65;
2900         }
2901         return L == 0;
2902
2903     case 0xC4: // DHT - define huffman table
2904         L = stbi__get16be(z->s) - 2;
2905         while (L > 0) {
2906             stbi_uc *v;
2907             int sizes[16], i, n = 0;
2908             int q = stbi__get8(z->s);
2909             int tc = q >> 4;
2910             int th = q & 15;
2911             if (tc > 1 || th > 3) return stbi__err("bad DHT header", "Corrupt JPEG");
2912             for (i = 0; i < 16; ++i) {
2913                 sizes[i] = stbi__get8(z->s);
2914                 n += sizes[i];
2915             }
2916             L -= 17;
2917             if (tc == 0) {
2918                 if (!stbi__build_huffman(z->huff_dc + th, sizes)) return 0;
2919                 v = z->huff_dc[th].values;
2920             }
2921             else {
2922                 if (!stbi__build_huffman(z->huff_ac + th, sizes)) return 0;
2923                 v = z->huff_ac[th].values;
2924             }
2925             for (i = 0; i < n; ++i)
2926                 v[i] = stbi__get8(z->s);
2927             if (tc != 0)
2928                 stbi__build_fast_ac(z->fast_ac[th], z->huff_ac + th);
2929             L -= n;
2930         }
2931         return L == 0;
2932     }
2933     // check for comment block or APP blocks
2934     if ((m >= 0xE0 && m <= 0xEF) || m == 0xFE) {
2935         stbi__skip(z->s, stbi__get16be(z->s) - 2);
2936         return 1;
2937     }
2938     return 0;
2939 }
2940
2941 // after we see SOS
2942 static int stbi__process_scan_header(stbi__jpeg *z)
2943 {
2944     int i;
2945     int Ls = stbi__get16be(z->s);
2946     z->scan_n = stbi__get8(z->s);
2947     if (z->scan_n < 1 || z->scan_n > 4 || z->scan_n > (int)z->s->img_n) return stbi__err("bad SOS
2948     component count", "Corrupt JPEG");
2949     if (Ls != 6 + 2 * z->scan_n) return stbi__err("bad SOS len", "Corrupt JPEG");
2950     for (i = 0; i < z->scan_n; ++i) {
2951         int id = stbi__get8(z->s), which;
2952         int q = stbi__get8(z->s);
2953         for (which = 0; which < z->s->img_n; ++which)
2954             if (z->img_comp[which].id == id)
2955                 break;

```

```

2955         if (which == z->s->img_n) return 0; // no match
2956         z->img_comp[which].hd = q >> 4; if (z->img_comp[which].hd > 3) return stbi__err("bad DC huff",
"Corrupt JPEG");
2957         z->img_comp[which].ha = q & 15; if (z->img_comp[which].ha > 3) return stbi__err("bad AC
huff", "Corrupt JPEG");
2958         z->order[i] = which;
2959     }
2960
2961     {
2962         int aa;
2963         z->spec_start = stbi__get8(z->s);
2964         z->spec_end = stbi__get8(z->s); // should be 63, but might be 0
2965         aa = stbi__get8(z->s);
2966         z->succ_high = (aa >> 4);
2967         z->succ_low = (aa & 15);
2968         if (z->progressive) {
2969             if (z->spec_start > 63 || z->spec_end > 63 || z->spec_start > z->spec_end || z->succ_high >
13 || z->succ_low > 13)
2970                 return stbi__err("bad SOS", "Corrupt JPEG");
2971         }
2972         else {
2973             if (z->spec_start != 0) return stbi__err("bad SOS", "Corrupt JPEG");
2974             if (z->succ_high != 0 || z->succ_low != 0) return stbi__err("bad SOS", "Corrupt JPEG");
2975             z->spec_end = 63;
2976         }
2977     }
2978
2979     return 1;
2980 }
2981
2982 static int stbi__free_jpeg_components(stbi__jpeg *z, int ncomp, int why)
2983 {
2984     int i;
2985     for (i = 0; i < ncomp; ++i) {
2986         if (z->img_comp[i].raw_data) {
2987             STBI_FREE(z->img_comp[i].raw_data);
2988             z->img_comp[i].raw_data = NULL;
2989             z->img_comp[i].data = NULL;
2990         }
2991         if (z->img_comp[i].raw_coeff) {
2992             STBI_FREE(z->img_comp[i].raw_coeff);
2993             z->img_comp[i].raw_coeff = 0;
2994             z->img_comp[i].coeff = 0;
2995         }
2996         if (z->img_comp[i].linebuf) {
2997             STBI_FREE(z->img_comp[i].linebuf);
2998             z->img_comp[i].linebuf = NULL;
2999         }
3000     }
3001     return why;
3002 }
3003
3004 static int stbi__process_frame_header(stbi__jpeg *z, int scan)
3005 {
3006     stbi__context *s = z->s;
3007     int Lf, p, i, q, h_max = 1, v_max = 1, c;
3008     Lf = stbi__get16be(s); if (Lf < 11) return stbi__err("bad SOF len", "Corrupt JPEG"); //
JPEG
3009     p = stbi__get8(s); if (p != 8) return stbi__err("only 8-bit", "JPEG format not
supported: 8-bit only"); // JPEG baseline
3010     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
3011     s->img_x = stbi__get16be(s); if (s->img_x == 0) return stbi__err("0 width", "Corrupt JPEG"); //
JPEG requires
3012     c = stbi__get8(s);
3013     if (c != 3 && c != 1) return stbi__err("bad component count", "Corrupt JPEG"); // JFIF requires
3014     s->img_n = c;
3015     for (i = 0; i < c; ++i) {
3016         z->img_comp[i].data = NULL;
3017         z->img_comp[i].linebuf = NULL;
3018     }
3019
3020     if (Lf != 8 + 3 * s->img_n) return stbi__err("bad SOF len", "Corrupt JPEG");
3021
3022     z->rgb = 0;
3023     for (i = 0; i < s->img_n; ++i) {
3024         static unsigned char rgb[3] = { 'R', 'G', 'B' };
3025         z->img_comp[i].id = stbi__get8(s);
3026         if (z->img_comp[i].id != i + 1) // JFIF requires
3027             if (z->img_comp[i].id != i) { // some version of jpegtran outputs non-JFIF-compliant
files!
3028                 // somethings output this (see
http://fileformats.archiveteam.org/wiki/JPEG#Color\_format)
3029                 if (z->img_comp[i].id != rgb[i])
3030                     return stbi__err("bad component ID", "Corrupt JPEG");
3031                 ++z->rgb;
3032     }

```

```

3033     q = stbi__get8(s);
3034     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");
3035     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");
3036     z->img_comp[i].tq = stbi__get8(s); if (z->img_comp[i].tq > 3) return stbi__err("bad TQ",
"Corrupt JPEG");
3037 }
3038
3039 if (scan != STBI__SCAN_load) return 1;
3040
3041 if (!stbi__mad3sizes_valid(s->img_x, s->img_y, s->img_n, 0)) return stbi__err("too large", "Image
too large to decode");
3042
3043 for (i = 0; i < s->img_n; ++i) {
3044     if (z->img_comp[i].h > h_max) h_max = z->img_comp[i].h;
3045     if (z->img_comp[i].v > v_max) v_max = z->img_comp[i].v;
3046 }
3047
3048 // compute interleaved mcu info
3049 z->img_h_max = h_max;
3050 z->img_v_max = v_max;
3051 z->img_mcu_w = h_max * 8;
3052 z->img_mcu_h = v_max * 8;
3053 // these sizes can't be more than 17 bits
3054 z->img_mcu_x = (s->img_x + z->img_mcu_w - 1) / z->img_mcu_w;
3055 z->img_mcu_y = (s->img_y + z->img_mcu_h - 1) / z->img_mcu_h;
3056
3057 for (i = 0; i < s->img_n; ++i) {
3058     // number of effective pixels (e.g. for non-interleaved MCU)
3059     z->img_comp[i].x = (s->img_x * z->img_comp[i].h + h_max - 1) / h_max;
3060     z->img_comp[i].y = (s->img_y * z->img_comp[i].v + v_max - 1) / v_max;
3061     // to simplify generation, we'll allocate enough memory to decode
3062     // the bogus oversized data from using interleaved MCUs and their
3063     // big blocks (e.g. a 16x16 iMCU on an image of width 33); we won't
3064     // discard the extra data until colorspace conversion
3065     //
3066     // img_mcu_x, img_mcu_y: <=17 bits; comp[i].h and .v are <=4 (checked earlier)
3067     // so these mults can't overflow with 32-bit ints (which we require)
3068     z->img_comp[i].w2 = z->img_mcu_x * z->img_comp[i].h * 8;
3069     z->img_comp[i].h2 = z->img_mcu_y * z->img_comp[i].v * 8;
3070     z->img_comp[i].coeff = 0;
3071     z->img_comp[i].raw_coeff = 0;
3072     z->img_comp[i].linebuf = NULL;
3073     z->img_comp[i].raw_data = stbi__malloc_mad2(z->img_comp[i].w2, z->img_comp[i].h2, 15);
3074     if (z->img_comp[i].raw_data == NULL)
3075         return stbi__free_jpeg_components(z, i + 1, stbi__err("outofmem", "Out of memory"));
3076     // align blocks for idct using mmx/sse
3077     z->img_comp[i].data = (stbi_uc*)((size_t)z->img_comp[i].raw_data + 15) & ~15;
3078     if (z->progressive) {
3079         // w2, h2 are multiples of 8 (see above)
3080         z->img_comp[i].coeff_w = z->img_comp[i].w2 / 8;
3081         z->img_comp[i].coeff_h = z->img_comp[i].h2 / 8;
3082         z->img_comp[i].raw_coeff = stbi__malloc_mad3(z->img_comp[i].w2, z->img_comp[i].h2,
sizeof(short), 15);
3083         if (z->img_comp[i].raw_coeff == NULL)
3084             return stbi__free_jpeg_components(z, i + 1, stbi__err("outofmem", "Out of memory"));
3085         z->img_comp[i].coeff = (short*)((size_t)z->img_comp[i].raw_coeff + 15) & ~15;
3086     }
3087 }
3088
3089 return 1;
3090 }
3091
3092 // use comparisons since in some cases we handle more than one case (e.g. SOF)
3093 #define stbi__DNL(x) ((x) == 0xdc)
3094 #define stbi__SOI(x) ((x) == 0xd8)
3095 #define stbi__EOI(x) ((x) == 0xd9)
3096 #define stbi__SOF(x) ((x) == 0xc0 || (x) == 0xc1 || (x) == 0xc2)
3097 #define stbi__SOS(x) ((x) == 0xda)
3098
3099 #define stbi__SOF_progressive(x) ((x) == 0xc2)
3100
3101 static int stbi__decode_jpeg_header(stbi__jpeg *z, int scan)
3102 {
3103     int m;
3104     z->marker = STBI__MARKER_none; // initialize cached marker to empty
3105     m = stbi__get_marker(z);
3106     if (!stbi__SOI(m)) return stbi__err("no SOI", "Corrupt JPEG");
3107     if (scan == STBI__SCAN_type) return 1;
3108     m = stbi__get_marker(z);
3109     while (!stbi__SOF(m)) {
3110         if (!stbi__process_marker(z, m)) return 0;
3111         m = stbi__get_marker(z);
3112         while (m == STBI__MARKER_none) {
3113             // some files have extra padding after their blocks, so ok, we'll scan
3114             if (stbi__at_eof(z->s)) return stbi__err("no SOF", "Corrupt JPEG");

```



```

3115         m = stbi__get_marker(z);
3116     }
3117 }
3118 z->progressive = stbi__SOF_progressive(m);
3119 if (!stbi__process_frame_header(z, scan)) return 0;
3120 return 1;
3121 }
3122
3123 // decode image to YCbCr format
3124 static int stbi__decode_jpeg_image(stbi__jpeg *j)
3125 {
3126     int m;
3127     for (m = 0; m < 4; m++) {
3128         j->img_comp[m].raw_data = NULL;
3129         j->img_comp[m].raw_coeff = NULL;
3130     }
3131     j->restart_interval = 0;
3132     if (!stbi__decode_jpeg_header(j, STBI__SCAN_load)) return 0;
3133     m = stbi__get_marker(j);
3134     while (!stbi__EOI(m)) {
3135         if (stbi__SOS(m)) {
3136             if (!stbi__process_scan_header(j)) return 0;
3137             if (!stbi__parse_entropy_coded_data(j)) return 0;
3138             if (j->marker == STBI__MARKER_none) {
3139                 // handle 0s at the end of image data from IP Kamera 9060
3140                 while (!stbi__at_eof(j->s)) {
3141                     int x = stbi__get8(j->s);
3142                     if (x == 255) {
3143                         j->marker = stbi__get8(j->s);
3144                         break;
3145                     }
3146                     else if (x != 0) {
3147                         return stbi__err("junk before marker", "Corrupt JPEG");
3148                     }
3149                 }
3150                 // if we reach eof without hitting a marker, stbi__get_marker() below will fail and
3151                 // we'll eventually return 0
3152             }
3153             else {
3154                 if (!stbi__process_marker(j, m)) return 0;
3155             }
3156             m = stbi__get_marker(j);
3157         }
3158         if (j->progressive)
3159             stbi__jpeg_finish(j);
3160     }
3161     return 1;
3162 }
3163 // static jfif-centered resampling (across block boundaries)
3164
3165 typedef stbi_uc *(*resample_row_func)(stbi_uc *out, stbi_uc *in0, stbi_uc *in1,
3166     int w, int hs);
3167
3168 #define stbi__div4(x) ((stbi_uc) ((x) >> 2))
3169
3170 static stbi_uc *resample_row_1(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
3171 {
3172     STBI_NOTUSED(out);
3173     STBI_NOTUSED(in_far);
3174     STBI_NOTUSED(w);
3175     STBI_NOTUSED(hs);
3176     return in_near;
3177 }
3178
3179 static stbi_uc* stbi__resample_row_v_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
3180 {
3181     // need to generate two samples vertically for every one in input
3182     int i;
3183     STBI_NOTUSED(hs);
3184     for (i = 0; i < w; ++i)
3185         out[i] = stbi__div4(3 * in_near[i] + in_far[i] + 2);
3186     return out;
3187 }
3188
3189 static stbi_uc* stbi__resample_row_h_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
3190 {
3191     // need to generate two samples horizontally for every one in input
3192     int i;
3193     stbi_uc *input = in_near;
3194
3195     if (w == 1) {
3196         // if only one sample, can't do any interpolation
3197         out[0] = out[1] = input[0];
3198         return out;
3199     }
3200 }

```

```

3201     out[0] = input[0];
3202     out[1] = stbi__div4(input[0] * 3 + input[1] + 2);
3203     for (i = 1; i < w - 1; ++i) {
3204         int n = 3 * input[i] + 2;
3205         out[i * 2 + 0] = stbi__div4(n + input[i - 1]);
3206         out[i * 2 + 1] = stbi__div4(n + input[i + 1]);
3207     }
3208     out[i * 2 + 0] = stbi__div4(input[w - 2] * 3 + input[w - 1] + 2);
3209     out[i * 2 + 1] = input[w - 1];
3210
3211     STBI_NOTUSED(in_far);
3212     STBI_NOTUSED(hs);
3213
3214     return out;
3215 }
3216
3217 #define stbi__div16(x) ((stbi_uc) ((x) >> 4))
3218
3219 static stbi_uc *stbi_resample_row_hv_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
3220 {
3221     // need to generate 2x2 samples for every one in input
3222     int i, t0, t1;
3223     if (w == 1) {
3224         out[0] = out[1] = stbi__div4(3 * in_near[0] + in_far[0] + 2);
3225         return out;
3226     }
3227
3228     t1 = 3 * in_near[0] + in_far[0];
3229     out[0] = stbi__div4(t1 + 2);
3230     for (i = 1; i < w; ++i) {
3231         t0 = t1;
3232         t1 = 3 * in_near[i] + in_far[i];
3233         out[i * 2 - 1] = stbi__div16(3 * t0 + t1 + 8);
3234         out[i * 2] = stbi__div16(3 * t1 + t0 + 8);
3235     }
3236     out[w * 2 - 1] = stbi__div4(t1 + 2);
3237
3238     STBI_NOTUSED(hs);
3239
3240     return out;
3241 }
3242
3243 #if defined(STBI_SSE2) || defined(STBI_NEON)
3244 static stbi_uc *stbi_resample_row_hv_2_simd(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w,
3245     int hs)
3246 {
3247     // need to generate 2x2 samples for every one in input
3248     int i = 0, t0, t1;
3249
3250     if (w == 1) {
3251         out[0] = out[1] = stbi__div4(3 * in_near[0] + in_far[0] + 2);
3252         return out;
3253     }
3254
3255     t1 = 3 * in_near[0] + in_far[0];
3256     // process groups of 8 pixels for as long as we can.
3257     // note we can't handle the last pixel in a row in this loop
3258     // because we need to handle the filter boundary conditions.
3259     for (; i < ((w - 1) & ~7); i += 8) {
3260         #if defined(STBI_SSE2)
3261             // load and perform the vertical filtering pass
3262             // this uses 3*x + y = 4*x + (y - x)
3263             __m128i zero = _mm_setzero_si128();
3264             __m128i farb = _mm_loadl_epi64((__m128i *) (in_far + i));
3265             __m128i nearb = _mm_loadl_epi64((__m128i *) (in_near + i));
3266             __m128i farw = _mm_unpacklo_epi8(farb, zero);
3267             __m128i nearw = _mm_unpacklo_epi8(nearb, zero);
3268             __m128i diff = _mm_sub_epi16(farw, nearw);
3269             __m128i nears = _mm_slli_epi16(nearw, 2);
3270             __m128i curr = _mm_add_epi16(nears, diff); // current row
3271
3272             // horizontal filter works the same based on shifted
3273             // row. "prev" is current row shifted right by 1
3274             // insert the previous pixel value (from t1).
3275             // "next" is current row shifted left by 1 pixel,
3276             // of next block of 8 pixels added in.
3277             __m128i prv0 = _mm_slli_si128(curr, 2);
3278             __m128i nxt0 = _mm_srli_si128(curr, 2);
3279             __m128i prev = _mm_insert_epi16(prv0, t1, 0);
3280             __m128i next = _mm_insert_epi16(nxt0, 3 * in_near[i + 8] + in_far[i + 8], 7);
3281
3282             // horizontal filter, polyphase implementation since it's convenient:
3283             // even pixels = 3*cur + prev = cur*4 + (prev - cur)
3284             // odd  pixels = 3*cur + next = cur*4 + (next - cur)

```

```

3284         // note the shared term.
3285         __m128i bias = _mm_set1_epi16(8);
3286         __m128i curs = _mm_slli_epi16(curr, 2);
3287         __m128i prvd = _mm_sub_epi16(prev, curr);
3288         __m128i nxd = _mm_sub_epi16(next, curr);
3289         __m128i curb = _mm_add_epi16(curs, bias);
3290         __m128i even = _mm_add_epi16(prvd, curb);
3291         __m128i odd = _mm_add_epi16(nxd, curb);
3292
3293         // interleave even and odd pixels, then undo scaling.
3294         __m128i int0 = _mm_unpacklo_epi16(even, odd);
3295         __m128i int1 = _mm_unpackhi_epi16(even, odd);
3296         __m128i de0 = _mm_srli_epi16(int0, 4);
3297         __m128i de1 = _mm_srli_epi16(int1, 4);
3298
3299         // pack and write output
3300         __m128i outv = _mm_packus_epi16(de0, de1);
3301         _mm_storeu_si128((__m128i *) (out + i * 2), outv);
3302 #elif defined(STBI_NEON)
3303         // load and perform the vertical filtering pass
3304         // this uses 3*x + y = 4*x + (y - x)
3305         uint8x8_t farb = vld1_u8(in_far + i);
3306         uint8x8_t nearb = vld1_u8(in_near + i);
3307         int16x8_t diff = vreinterpretq_s16_u16(vsubl_u8(farb, nearb));
3308         int16x8_t nears = vreinterpretq_s16_u16(vshll_n_u8(nearb, 2));
3309         int16x8_t curr = vaddq_s16(nears, diff); // current row
3310
3311         // horizontal filter works the same based on shifted
3312         // row. "prev" is current row shifted right by 1
3313         // pixel; we need to
3314         // insert the previous pixel value (from t1).
3315         // "next" is current row shifted left by 1 pixel, with
3316         // of next block of 8 pixels added in.
3317         int16x8_t prv0 = vextq_s16(curr, curr, 7);
3318         int16x8_t nxt0 = vextq_s16(curr, curr, 1);
3319         int16x8_t prev = vsetq_lane_s16(t1, prv0, 0);
3320         int16x8_t next = vsetq_lane_s16(3 * in_near[i + 8] + in_far[i + 8], nxt0, 7);
3321
3322         // horizontal filter, polyphase implementation since it's convenient:
3323         // even pixels = 3*cur + prev = cur*4 + (prev - cur)
3324         // odd pixels = 3*cur + next = cur*4 + (next - cur)
3325         // note the shared term.
3326         int16x8_t curs = vshlq_n_s16(curr, 2);
3327         int16x8_t prvd = vsubq_s16(prev, curs);
3328         int16x8_t nxd = vsubq_s16(next, curs);
3329         int16x8_t even = vaddq_s16(curs, prvd);
3330         int16x8_t odd = vaddq_s16(curs, nxd);
3331
3332         // undo scaling and round, then store with even/odd phases interleaved
3333         uint8x8x2_t o;
3334         o.val[0] = vqsrshr_n_s16(even, 4);
3335         o.val[1] = vqsrshr_n_s16(odd, 4);
3336         vst2_u8(out + i * 2, o);
3337 #endif
3338         // "previous" value for next iter
3339         t1 = 3 * in_near[i + 7] + in_far[i + 7];
3340     }
3341
3342     t0 = t1;
3343     t1 = 3 * in_near[i] + in_far[i];
3344     out[i * 2] = stbi_div16(3 * t1 + t0 + 8);
3345
3346     for (++i; i < w; ++i) {
3347         t0 = t1;
3348         t1 = 3 * in_near[i] + in_far[i];
3349         out[i * 2 - 1] = stbi_div16(3 * t0 + t1 + 8);
3350         out[i * 2] = stbi_div16(3 * t1 + t0 + 8);
3351     }
3352     out[w * 2 - 1] = stbi_div4(t1 + 2);
3353
3354     STBI_NOTUSED(hs);
3355
3356     return out;
3357 }
3358 #endif
3359
3360 static stbi_uc *stbi__resample_row_generic(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int
hs)
3361 {
3362     // resample with nearest-neighbor
3363     int i, j;
3364     STBI_NOTUSED(in_far);
3365     for (i = 0; i < w; ++i)
3366         for (j = 0; j < hs; ++j)

```

```

3367         out[i*hs + j] = in_near[i];
3368     return out;
3369 }
3370
3371 #ifdef STBI_JPEG_OLD
3372 // this is the same YCbCr-to-RGB calculation that stb_image has used
3373 // historically before the algorithm changes in 1.49
3374 #define float2fixed(x) ((int) ((x) * 65536 + 0.5))
3375 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)
3376 {
3377     int i;
3378     for (i = 0; i < count; ++i) {
3379         int y_fixed = (y[i] << 16) + 32768; // rounding
3380         int r, g, b;
3381         int cr = pcr[i] - 128;
3382         int cb = pcb[i] - 128;
3383         r = y_fixed + cr*float2fixed(1.40200f);
3384         g = y_fixed - cr*float2fixed(0.71414f) - cb*float2fixed(0.34414f);
3385         b = y_fixed + cb*float2fixed(1.77200f);
3386         r >>= 16;
3387         g >>= 16;
3388         b >>= 16;
3389         if ((unsigned)r > 255) { if (r < 0) r = 0; else r = 255; }
3390         if ((unsigned)g > 255) { if (g < 0) g = 0; else g = 255; }
3391         if ((unsigned)b > 255) { if (b < 0) b = 0; else b = 255; }
3392         out[0] = (stbi_uc)r;
3393         out[1] = (stbi_uc)g;
3394         out[2] = (stbi_uc)b;
3395         out[3] = 255;
3396         out += step;
3397     }
3398 }
3399 #else
3400 // this is a reduced-precision calculation of YCbCr-to-RGB introduced
3401 // to make sure the code produces the same results in both SIMD and scalar
3402 #define float2fixed(x) (((int) ((x) * 4096.0f + 0.5f)) << 8)
3403 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)
3404 {
3405     int i;
3406     for (i = 0; i < count; ++i) {
3407         int y_fixed = (y[i] << 20) + (1 << 19); // rounding
3408         int r, g, b;
3409         int cr = pcr[i] - 128;
3410         int cb = pcb[i] - 128;
3411         r = y_fixed + cr* float2fixed(1.40200f);
3412         g = y_fixed + (cr*-float2fixed(0.71414f)) + ((cb*-float2fixed(0.34414f)) & 0xffff0000);
3413         b = y_fixed + cb* float2fixed(1.77200f);
3414         r >>= 20;
3415         g >>= 20;
3416         b >>= 20;
3417         if ((unsigned)r > 255) { if (r < 0) r = 0; else r = 255; }
3418         if ((unsigned)g > 255) { if (g < 0) g = 0; else g = 255; }
3419         if ((unsigned)b > 255) { if (b < 0) b = 0; else b = 255; }
3420         out[0] = (stbi_uc)r;
3421         out[1] = (stbi_uc)g;
3422         out[2] = (stbi_uc)b;
3423         out[3] = 255;
3424         out += step;
3425     }
3426 }
3427 #endif
3428
3429 #if defined(STBI_SSE2) || defined(STBI_NEON)
3430 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)
3431 {
3432     int i = 0;
3433
3434 #ifdef STBI_SSE2
3435     // step == 3 is pretty ugly on the final interleave, and i'm not convinced
3436     // it's useful in practice (you wouldn't use it for textures, for example).
3437     // so just accelerate step == 4 case.
3438     if (step == 4) {
3439         // this is a fairly straightforward implementation and not super-optimized.
3440         __m128i signflip = _mm_set1_epi8(-0x80);
3441         __m128i cr_const0 = _mm_set1_epi16((short) (1.40200f*4096.0f + 0.5f));
3442         __m128i cr_const1 = _mm_set1_epi16(-(short) (0.71414f*4096.0f + 0.5f));
3443         __m128i cb_const0 = _mm_set1_epi16(-(short) (0.34414f*4096.0f + 0.5f));
3444         __m128i cb_const1 = _mm_set1_epi16((short) (1.77200f*4096.0f + 0.5f));
3445         __m128i y_bias = _mm_set1_epi8((char) (unsigned char) 128);
3446         __m128i xw = _mm_set1_epi16(255); // alpha channel
3447
3448         for (; i + 7 < count; i += 8) {
3449             // load
3450             __m128i y_bytes = _mm_loadl_epi64((__m128i *) (y + i));

```

```

3451     __m128i cr_bytes = _mm_loadl_epi64((__m128i *) (pcr + i));
3452     __m128i cb_bytes = _mm_loadl_epi64((__m128i *) (pcb + i));
3453     __m128i cr_biased = _mm_xor_si128(cr_bytes, signflip); // -128
3454     __m128i cb_biased = _mm_xor_si128(cb_bytes, signflip); // -128
3455
3456                                     // unpack to short (and left-shift
cr, cb by 8)
3457     __m128i yw = _mm_unpacklo_epi8(y_bias, y_bytes);
3458     __m128i crw = _mm_unpacklo_epi8(_mm_setzero_si128(), cr_biased);
3459     __m128i cbw = _mm_unpacklo_epi8(_mm_setzero_si128(), cb_biased);
3460
3461     // color transform
3462     __m128i yws = _mm_srli_epi16(yw, 4);
3463     __m128i cr0 = _mm_mulhi_epi16(cr_const0, crw);
3464     __m128i cb0 = _mm_mulhi_epi16(cb_const0, cbw);
3465     __m128i cb1 = _mm_mulhi_epi16(cbw, cb_const1);
3466     __m128i cr1 = _mm_mulhi_epi16(crw, cr_const1);
3467     __m128i rws = _mm_add_epi16(cr0, yws);
3468     __m128i gwt = _mm_add_epi16(cb0, yws);
3469     __m128i bws = _mm_add_epi16(yws, cb1);
3470     __m128i gws = _mm_add_epi16(gwt, cr1);
3471
3472     // descale
3473     __m128i rw = _mm_srai_epi16(rws, 4);
3474     __m128i bw = _mm_srai_epi16(bws, 4);
3475     __m128i gw = _mm_srai_epi16(gws, 4);
3476
3477     // back to byte, set up for transpose
3478     __m128i brb = _mm_packus_epi16(rw, bw);
3479     __m128i gxb = _mm_packus_epi16(gw, xw);
3480
3481     // transpose to interleave channels
3482     __m128i t0 = _mm_unpacklo_epi8(brb, gxb);
3483     __m128i t1 = _mm_unpackhi_epi8(brb, gxb);
3484     __m128i o0 = _mm_unpacklo_epi16(t0, t1);
3485     __m128i o1 = _mm_unpackhi_epi16(t0, t1);
3486
3487     // store
3488     __mm_storeu_si128((__m128i *) (out + 0), o0);
3489     __mm_storeu_si128((__m128i *) (out + 16), o1);
3490     out += 32;
3491 }
3492 }
3493 #endif
3494
3495 #ifdef STBI_NEON
3496     // in this version, step=3 support would be easy to add. but is there demand?
3497     if (step == 4) {
3498         // this is a fairly straightforward implementation and not super-optimized.
3499         uint8x8_t signflip = vdup_n_u8(0x80);
3500         int16x8_t cr_const0 = vdupq_n_s16((short)(1.40200f*4096.0f + 0.5f));
3501         int16x8_t cr_const1 = vdupq_n_s16(-(short)(0.71414f*4096.0f + 0.5f));
3502         int16x8_t cb_const0 = vdupq_n_s16(-(short)(0.34414f*4096.0f + 0.5f));
3503         int16x8_t cb_const1 = vdupq_n_s16((short)(1.77200f*4096.0f + 0.5f));
3504
3505         for (; i + 7 < count; i += 8) {
3506             // load
3507             uint8x8_t y_bytes = vld1_u8(y + i);
3508             uint8x8_t cr_bytes = vld1_u8(pcr + i);
3509             uint8x8_t cb_bytes = vld1_u8(pcb + i);
3510             int8x8_t cr_biased = vreinterpret_s8_u8(vsub_u8(cr_bytes, signflip));
3511             int8x8_t cb_biased = vreinterpret_s8_u8(vsub_u8(cb_bytes, signflip));
3512
3513             // expand to s16
3514             int16x8_t yws = vreinterpretq_s16_u16(vshll_n_u8(y_bytes, 4));
3515             int16x8_t crw = vshll_n_s8(cr_biased, 7);
3516             int16x8_t cbw = vshll_n_s8(cb_biased, 7);
3517
3518             // color transform
3519             int16x8_t cr0 = vqdmulhq_s16(crw, cr_const0);
3520             int16x8_t cb0 = vqdmulhq_s16(cbw, cb_const0);
3521             int16x8_t cr1 = vqdmulhq_s16(crw, cr_const1);
3522             int16x8_t cb1 = vqdmulhq_s16(cbw, cb_const1);
3523             int16x8_t rws = vaddq_s16(yws, cr0);
3524             int16x8_t gws = vaddq_s16(vaddq_s16(yws, cb0), cr1);
3525             int16x8_t bws = vaddq_s16(yws, cb1);
3526
3527             // undo scaling, round, convert to byte
3528             uint8x8x4_t o;
3529             o.val[0] = vqshrshrn_n_s16(rws, 4);
3530             o.val[1] = vqshrshrn_n_s16(gws, 4);
3531             o.val[2] = vqshrshrn_n_s16(bws, 4);
3532             o.val[3] = vdup_n_u8(255);
3533
3534             // store, interleaving r/g/b/a
3535             vst4_u8(out, o);
3536             out += 8 * 4;

```

```

3537     }
3538 }
3539 #endif
3540
3541     for (; i < count; ++i) {
3542         int y_fixed = (y[i] << 20) + (1 << 19); // rounding
3543         int r, g, b;
3544         int cr = pcr[i] - 128;
3545         int cb = pcb[i] - 128;
3546         r = y_fixed + cr* float2fixed(1.40200f);
3547         g = y_fixed + cr*-float2fixed(0.71414f) + ((cb*-float2fixed(0.34414f)) & 0xffff0000);
3548         b = y_fixed + cb* float2fixed(1.77200f);
3549         r >>= 20;
3550         g >>= 20;
3551         b >>= 20;
3552         if ((unsigned)r > 255) { if (r < 0) r = 0; else r = 255; }
3553         if ((unsigned)g > 255) { if (g < 0) g = 0; else g = 255; }
3554         if ((unsigned)b > 255) { if (b < 0) b = 0; else b = 255; }
3555         out[0] = (stbi_uc)r;
3556         out[1] = (stbi_uc)g;
3557         out[2] = (stbi_uc)b;
3558         out[3] = 255;
3559         out += step;
3560     }
3561 }
3562 #endif
3563
3564 // set up the kernels
3565 static void stbi__setup_jpeg(stbi__jpeg *j)
3566 {
3567     j->idct_block_kernel = stbi__idct_block;
3568     j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_row;
3569     j->resample_row_hv_2_kernel = stbi__resample_row_hv_2;
3570
3571     #ifdef STBI_SSE2
3572     if (stbi__sse2_available()) {
3573         j->idct_block_kernel = stbi__idct_simd;
3574     }
3575     #ifndef STBI_JPEG_OLD
3576     j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_simd;
3577     j->resample_row_hv_2_kernel = stbi__resample_row_hv_2_simd;
3578     }
3579     #endif
3580
3581     #ifdef STBI_NEON
3582     j->idct_block_kernel = stbi__idct_simd;
3583     #ifndef STBI_JPEG_OLD
3584     j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_simd;
3585     j->resample_row_hv_2_kernel = stbi__resample_row_hv_2_simd;
3586     }
3587     #endif
3588 }
3589
3590 // clean up the temporary component buffers
3591 static void stbi__cleanup_jpeg(stbi__jpeg *j)
3592 {
3593     stbi__free_jpeg_components(j, j->s->img_n, 0);
3594 }
3595
3596 typedef struct
3597 {
3598     resample_row_func resample;
3599     stbi_uc *line0, *line1;
3600     int hs, vs; // expansion factor in each axis
3601     int w_lores; // horizontal pixels pre-expansion
3602     int ystep; // how far through vertical expansion we are
3603     int ypos; // which pre-expansion row we're on
3604 } stbi__resample;
3605
3606 static stbi_uc *load_jpeg_image(stbi__jpeg *z, int *out_x, int *out_y, int *comp, int req_comp)
3607 {
3608     int n, decode_n;
3609     z->s->img_n = 0; // make stbi__cleanup_jpeg safe
3610
3611     // validate req_comp
3612     if (req_comp < 0 || req_comp > 4) return stbi__errpuc("bad req_comp", "Internal error");
3613
3614     // load a jpeg image from whichever source, but leave in YCbCr format
3615     if (!stbi__decode_jpeg_image(z)) { stbi__cleanup_jpeg(z); return NULL; }
3616
3617     // determine actual number of components to generate
3618     n = req_comp ? req_comp : z->s->img_n;
3619
3620     if (z->s->img_n == 3 && n < 3)
3621         decode_n = 1;
3622     else
3623         decode_n = z->s->img_n;

```

```

3624
3625 // resample and color-convert
3626 {
3627     int k;
3628     unsigned int i, j;
3629     stbi_uc *output;
3630     stbi_uc *coutput[4];
3631
3632     stbi__resample res_comp[4];
3633
3634     for (k = 0; k < decode_n; ++k) {
3635         stbi__resample *r = &res_comp[k];
3636
3637         // allocate line buffer big enough for upsampling off the edges
3638         // with upsample factor of 4
3639         z->img_comp[k].linebuf = (stbi_uc *)stbi__malloc(z->s->img_x + 3);
3640         if (!z->img_comp[k].linebuf) { stbi__cleanup_jpeg(z); return stbi__errpuc("outofmem", "Out
of memory"); }
3641
3642         r->hs = z->img_h_max / z->img_comp[k].h;
3643         r->vs = z->img_v_max / z->img_comp[k].v;
3644         r->ystep = r->vs » 1;
3645         r->w_lores = (z->s->img_x + r->hs - 1) / r->hs;
3646         r->ypos = 0;
3647         r->line0 = r->line1 = z->img_comp[k].data;
3648
3649         if (r->hs == 1 && r->vs == 1) r->resample = resample_row_1;
3650         else if (r->hs == 1 && r->vs == 2) r->resample = stbi__resample_row_v_2;
3651         else if (r->hs == 2 && r->vs == 1) r->resample = stbi__resample_row_h_2;
3652         else if (r->hs == 2 && r->vs == 2) r->resample = z->resample_row_hv_2_kernel;
3653         else
3654             r->resample = stbi__resample_row_generic;
3655     }
3656
3657     // can't error after this so, this is safe
3658     output = (stbi_uc *)stbi__malloc_mad3(n, z->s->img_x, z->s->img_y, 1);
3659     if (!output) { stbi__cleanup_jpeg(z); return stbi__errpuc("outofmem", "Out of memory"); }
3660
3661     // now go ahead and resample
3662     for (j = 0; j < z->s->img_y; ++j) {
3663         stbi_uc *out = output + n * z->s->img_x * j;
3664         for (k = 0; k < decode_n; ++k) {
3665             stbi__resample *r = &res_comp[k];
3666             int y_bot = r->ystep >= (r->vs » 1);
3667             coutput[k] = r->resample(z->img_comp[k].linebuf,
3668                                     y_bot ? r->line1 : r->line0,
3669                                     y_bot ? r->line0 : r->line1,
3670                                     r->w_lores, r->hs);
3671             if (++r->ystep >= r->vs) {
3672                 r->ystep = 0;
3673                 r->line0 = r->line1;
3674                 if (++r->ypos < z->img_comp[k].y)
3675                     r->line1 += z->img_comp[k].w2;
3676             }
3677         }
3678         if (n >= 3) {
3679             stbi_uc *y = coutput[0];
3680             if (z->s->img_n == 3) {
3681                 if (z->rgb == 3) {
3682                     for (i = 0; i < z->s->img_x; ++i) {
3683                         out[0] = y[i];
3684                         out[1] = coutput[1][i];
3685                         out[2] = coutput[2][i];
3686                         out[3] = 255;
3687                         out += n;
3688                     }
3689                 }
3690                 else {
3691                     z->YCbCr_to_RGB_kernel(out, y, coutput[1], coutput[2], z->s->img_x, n);
3692                 }
3693             }
3694             else
3695                 for (i = 0; i < z->s->img_x; ++i) {
3696                     out[0] = out[1] = out[2] = y[i];
3697                     out[3] = 255; // not used if n==3
3698                     out += n;
3699                 }
3700         }
3701         else {
3702             stbi_uc *y = coutput[0];
3703             if (n == 1)
3704                 for (i = 0; i < z->s->img_x; ++i) out[i] = y[i];
3705             else
3706                 for (i = 0; i < z->s->img_x; ++i) *out++ = y[i], *out++ = 255;
3707         }
3708     }
3709     stbi__cleanup_jpeg(z);
3710     *out_x = z->s->img_x;

```

```

3710         *out_y = z->s->img_y;
3711         if (comp) *comp = z->s->img_n; // report original components, not output
3712         return output;
3713     }
3714 }
3715
3716 static void *stbi__jpeg_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
    stbi__result_info *ri)
3717 {
3718     unsigned char* result;
3719     stbi__jpeg* j = (stbi__jpeg*)stbi__malloc(sizeof(stbi__jpeg));
3720     j->s = s;
3721     stbi__setup_jpeg(j);
3722     result = load_jpeg_image(j, x, y, comp, req_comp);
3723     STBI_FREE(j);
3724     return result;
3725 }
3726
3727 static int stbi__jpeg_test(stbi__context *s)
3728 {
3729     int r;
3730     stbi__jpeg j;
3731     j.s = s;
3732     stbi__setup_jpeg(&j);
3733     r = stbi__decode_jpeg_header(&j, STBI__SCAN_type);
3734     stbi__rewind(s);
3735     return r;
3736 }
3737
3738 static int stbi__jpeg_info_raw(stbi__jpeg *j, int *x, int *y, int *comp)
3739 {
3740     if (!stbi__decode_jpeg_header(j, STBI__SCAN_header)) {
3741         stbi__rewind(j->s);
3742         return 0;
3743     }
3744     if (x) *x = j->s->img_x;
3745     if (y) *y = j->s->img_y;
3746     if (comp) *comp = j->s->img_n;
3747     return 1;
3748 }
3749
3750 static int stbi__jpeg_info(stbi__context *s, int *x, int *y, int *comp)
3751 {
3752     int result;
3753     stbi__jpeg* j = (stbi__jpeg*)(stbi__malloc(sizeof(stbi__jpeg)));
3754     j->s = s;
3755     result = stbi__jpeg_info_raw(j, x, y, comp);
3756     STBI_FREE(j);
3757     return result;
3758 }
3759 #endif
3760
3761 // public domain zlib decode    v0.2  Sean Barrett 2006-11-18
3762 //     simple implementation
3763 //     - all input must be provided in an upfront buffer
3764 //     - all output is written to a single output buffer (can malloc/realloc)
3765 //     performance
3766 //     - fast huffman
3767
3768 #ifndef STBI_NO_ZLIB
3769
3770 // fast-way is faster to check than jpeg huffman, but slow way is slower
3771 #define STBI__ZFAST_BITS 9 // accelerate all cases in default tables
3772 #define STBI__ZFAST_MASK ((1 < STBI__ZFAST_BITS) - 1)
3773
3774 // zlib-style huffman encoding
3775 // (jpegs packs from left, zlib from right, so can't share code)
3776 typedef struct
3777 {
3778     stbi__uint16 fast[1 < STBI__ZFAST_BITS];
3779     stbi__uint16 firstcode[16];
3780     int maxcode[17];
3781     stbi__uint16 firstsymbol[16];
3782     stbi_uc size[288];
3783     stbi__uint16 value[288];
3784 } stbi__zhuffman;
3785
3786 stbi_inline static int stbi__bitreverse16(int n)
3787 {
3788     n = ((n & 0xAAAA) >> 1) | ((n & 0x5555) << 1);
3789     n = ((n & 0xCCCC) >> 2) | ((n & 0x3333) << 2);
3790     n = ((n & 0xF0F0) >> 4) | ((n & 0x0F0F) << 4);
3791     n = ((n & 0xFF00) >> 8) | ((n & 0x00FF) << 8);
3792     return n;
3793 }
3794
3795 stbi_inline static int stbi__bit_reverse(int v, int bits)

```



```

3796 {
3797     STBI_ASSERT(bits <= 16);
3798     // to bit reverse n bits, reverse 16 and shift
3799     // e.g. 11 bits, bit reverse and shift away 5
3800     return stbi__bitreverse16(v) >> (16 - bits);
3801 }
3802
3803 static int stbi__zbuild_huffman(stbi__zhuffman *z, stbi_uc *sizelist, int num)
3804 {
3805     int i, k = 0;
3806     int code, next_code[16], sizes[17];
3807
3808     // DEFLATE spec for generating codes
3809     memset(sizes, 0, sizeof(sizes));
3810     memset(z->fast, 0, sizeof(z->fast));
3811     for (i = 0; i < num; ++i)
3812         ++sizes[sizelist[i]];
3813     sizes[0] = 0;
3814     for (i = 1; i < 16; ++i)
3815         if (sizes[i] > (1 << i))
3816             return stbi__err("bad sizes", "Corrupt PNG");
3817     code = 0;
3818     for (i = 1; i < 16; ++i) {
3819         next_code[i] = code;
3820         z->firstcode[i] = (stbi_uint16)code;
3821         z->firstsymbol[i] = (stbi_uint16)i;
3822         code = (code + sizes[i]);
3823         if (sizes[i])
3824             if (code - 1 >= (1 << i)) return stbi__err("bad codelengths", "Corrupt PNG");
3825         z->maxcode[i] = code << (16 - i); // preshift for inner loop
3826         code <= 1;
3827         k += sizes[i];
3828     }
3829     z->maxcode[16] = 0x10000; // sentinel
3830     for (i = 0; i < num; ++i) {
3831         int s = sizelist[i];
3832         if (s) {
3833             int c = next_code[s] - z->firstcode[s] + z->firstsymbol[s];
3834             stbi_uint16 fastv = (stbi_uint16)((s << 9) | i);
3835             z->size[c] = (stbi_uc)s;
3836             z->value[c] = (stbi_uint16)i;
3837             if (s <= STBI_ZFAST_BITS) {
3838                 int j = stbi__bit_reverse(next_code[s], s);
3839                 while (j < (1 << STBI_ZFAST_BITS)) {
3840                     z->fast[j] = fastv;
3841                     j += (1 << s);
3842                 }
3843             }
3844             ++next_code[s];
3845         }
3846     }
3847     return 1;
3848 }
3849
3850 // zlib-from-memory implementation for PNG reading
3851 //     because PNG allows splitting the zlib stream arbitrarily,
3852 //     and it's annoying structurally to have PNG call ZLIB call PNG,
3853 //     we require PNG read all the IDATs and combine them into a single
3854 //     memory buffer
3855
3856 typedef struct
3857 {
3858     stbi_uc *zbuffer, *zbuffer_end;
3859     int num_bits;
3860     stbi_uint32 code_buffer;
3861
3862     char *zout;
3863     char *zout_start;
3864     char *zout_end;
3865     int z_expandable;
3866
3867     stbi__zhuffman z_length, z_distance;
3868 } stbi__zbuf;
3869
3870 stbi_inline static stbi_uc stbi__zget8(stbi__zbuf *z)
3871 {
3872     if (z->zbuffer >= z->zbuffer_end) return 0;
3873     return *z->zbuffer++;
3874 }
3875
3876 static void stbi__fill_bits(stbi__zbuf *z)
3877 {
3878     do {
3879         STBI_ASSERT(z->code_buffer < (1U << z->num_bits));
3880         z->code_buffer |= (unsigned int)stbi__zget8(z) << z->num_bits;
3881         z->num_bits += 8;
3882     } while (z->num_bits <= 24);

```

```

3883 }
3884
3885 stbi_inline static unsigned int stbi__zreceive(stbi__zbuf *z, int n)
3886 {
3887     unsigned int k;
3888     if (z->num_bits < n) stbi__fill_bits(z);
3889     k = z->code_buffer & ((1 << n) - 1);
3890     z->code_buffer >>= n;
3891     z->num_bits -= n;
3892     return k;
3893 }
3894
3895 static int stbi__zhuffman_decode_slowpath(stbi__zbuf *a, stbi__zhuffman *z)
3896 {
3897     int b, s, k;
3898     // not resolved by fast table, so compute it the slow way
3899     // use jpeg approach, which requires MSbits at top
3900     k = stbi__bit_reverse(a->code_buffer, 16);
3901     for (s = STBI__ZFAST_BITS + 1; ; ++s)
3902         if (k < z->maxcode[s])
3903             break;
3904     if (s == 16) return -1; // invalid code!
3905     // code size is s, so:
3906     b = (k >> (16 - s)) - z->firstcode[s] + z->firstsymbol[s];
3907     STBI_ASSERT(z->size[b] == s);
3908     a->code_buffer >>= s;
3909     a->num_bits -= s;
3910     return z->value[b];
3911 }
3912
3913 stbi_inline static int stbi__zhuffman_decode(stbi__zbuf *a, stbi__zhuffman *z)
3914 {
3915     int b, s;
3916     if (a->num_bits < 16) stbi__fill_bits(a);
3917     b = z->fast[a->code_buffer & STBI__ZFAST_MASK];
3918     if (b) {
3919         s = b >> 9;
3920         a->code_buffer >>= s;
3921         a->num_bits -= s;
3922         return b & 511;
3923     }
3924     return stbi__zhuffman_decode_slowpath(a, z);
3925 }
3926
3927 static int stbi__zexpand(stbi__zbuf *z, char *zout, int n) // need to make room for n bytes
3928 {
3929     char *q;
3930     int cur, limit, old_limit;
3931     z->zout = zout;
3932     if (!z->z_expandable) return stbi__err("output buffer limit", "Corrupt PNG");
3933     cur = (int)(z->zout - z->zout_start);
3934     limit = old_limit = (int)(z->zout_end - z->zout_start);
3935     while (cur + n > limit)
3936         limit *= 2;
3937     q = (char *)STBI_REALLOC_SIZED(z->zout_start, old_limit, limit);
3938     STBI_NOTUSED(old_limit);
3939     if (q == NULL) return stbi__err("outofmem", "Out of memory");
3940     z->zout_start = q;
3941     z->zout = q + cur;
3942     z->zout_end = q + limit;
3943     return 1;
3944 }
3945
3946 static int stbi__zlength_base[31] = {
3947     3,4,5,6,7,8,9,10,11,13,
3948     15,17,19,23,27,31,35,43,51,59,
3949     67,83,99,115,131,163,195,227,258,0,0 };
3950
3951 static int stbi__zlength_extra[31] =
3952 { 0,0,0,0,0,0,0,1,1,1,1,2,2,2,3,3,3,4,4,4,5,5,5,0,0,0 };
3953
3954 static int stbi__zdist_base[32] = { 1,2,3,4,5,7,9,13,17,25,33,49,65,97,129,193,
3955 257,385,513,769,1025,1537,2049,3073,4097,6145,8193,12289,16385,24577,0,0 };
3956
3957 static int stbi__zdist_extra[32] =
3958 { 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 };
3959
3960 static int stbi__parse_huffman_block(stbi__zbuf *a)
3961 {
3962     char *zout = a->zout;
3963     for (;;) {
3964         int z = stbi__zhuffman_decode(a, &a->z_length);
3965         if (z < 256) {
3966             if (z < 0) return stbi__err("bad huffman code", "Corrupt PNG"); // error in huffman codes
3967             if (zout >= a->zout_end) {
3968                 if (!stbi__zexpand(a, zout, 1)) return 0;
3969                 zout = a->zout;

```

```

3970     }
3971     *zout++ = (char)z;
3972 }
3973 else {
3974     stbi_uc *p;
3975     int len, dist;
3976     if (z == 256) {
3977         a->zout = zout;
3978         return 1;
3979     }
3980     z -= 257;
3981     len = stbi__zlength_base[z];
3982     if (stbi__zlength_extra[z]) len += stbi__zreceive(a, stbi__zlength_extra[z]);
3983     z = stbi__zhuffman_decode(a, &a->z_distance);
3984     if (z < 0) return stbi__err("bad huffman code", "Corrupt PNG");
3985     dist = stbi__zdist_base[z];
3986     if (stbi__zdist_extra[z]) dist += stbi__zreceive(a, stbi__zdist_extra[z]);
3987     if (zout - a->zout_start < dist) return stbi__err("bad dist", "Corrupt PNG");
3988     if (zout + len > a->zout_end) {
3989         if (!stbi__zexpand(a, zout, len)) return 0;
3990         zout = a->zout;
3991     }
3992     p = (stbi_uc *) (zout - dist);
3993     if (dist == 1) { // run of one byte; common in images.
3994         stbi_uc v = *p;
3995         if (len) { do *zout++ = v; while (--len); }
3996     }
3997     else {
3998         if (len) { do *zout++ = *p++; while (--len); }
3999     }
4000 }
4001 }
4002 }
4003
4004 static int stbi__compute_huffman_codes(stbi__zbuf *a)
4005 {
4006     static stbi_uc length_dezigzag[19] = { 16,17,18,0,8,7,9,6,10,5,11,4,12,3,13,2,14,1,15 };
4007     stbi__zhuffman z_codelength;
4008     stbi_uc lencodes[286 + 32 + 137]; //padding for maximum single op
4009     stbi_uc codelength_sizes[19];
4010     int i, n;
4011
4012     int hlit = stbi__zreceive(a, 5) + 257;
4013     int hdist = stbi__zreceive(a, 5) + 1;
4014     int hclen = stbi__zreceive(a, 4) + 4;
4015     int ntot = hlit + hdist;
4016
4017     memset(codelength_sizes, 0, sizeof(codelength_sizes));
4018     for (i = 0; i < hclen; ++i) {
4019         int s = stbi__zreceive(a, 3);
4020         codelength_sizes[length_dezigzag[i]] = (stbi_uc)s;
4021     }
4022     if (!stbi__zbuild_huffman(&z_codelength, codelength_sizes, 19)) return 0;
4023
4024     n = 0;
4025     while (n < ntot) {
4026         int c = stbi__zhuffman_decode(a, &z_codelength);
4027         if (c < 0 || c >= 19) return stbi__err("bad codelengths", "Corrupt PNG");
4028         if (c < 16)
4029             lencodes[n++] = (stbi_uc)c;
4030         else {
4031             stbi_uc fill = 0;
4032             if (c == 16) {
4033                 c = stbi__zreceive(a, 2) + 3;
4034                 if (n == 0) return stbi__err("bad codelengths", "Corrupt PNG");
4035                 fill = lencodes[n - 1];
4036             }
4037             else if (c == 17)
4038                 c = stbi__zreceive(a, 3) + 3;
4039             else {
4040                 STBI_ASSERT(c == 18);
4041                 c = stbi__zreceive(a, 7) + 11;
4042             }
4043             if (ntot - n < c) return stbi__err("bad codelengths", "Corrupt PNG");
4044             memset(lencodes + n, fill, c);
4045             n += c;
4046         }
4047     }
4048     if (n != ntot) return stbi__err("bad codelengths", "Corrupt PNG");
4049     if (!stbi__zbuild_huffman(&a->z_length, lencodes, hlit)) return 0;
4050     if (!stbi__zbuild_huffman(&a->z_distance, lencodes + hlit, hdist)) return 0;
4051     return 1;
4052 }
4053
4054 static int stbi__parse_uncompressed_block(stbi__zbuf *a)
4055 {
4056     stbi_uc header[4];

```

```

4057     int len, nlen, k;
4058     if (a->num_bits & 7)
4059         stbi__zreceive(a, a->num_bits & 7); // discard
4060                                         // drain the bit-packed data into header
4061     k = 0;
4062     while (a->num_bits > 0) {
4063         header[k++] = (stbi_uc)(a->code_buffer & 255); // suppress MSVC run-time check
4064         a->code_buffer >>= 8;
4065         a->num_bits -= 8;
4066     }
4067     STBI_ASSERT(a->num_bits == 0);
4068     // now fill header the normal way
4069     while (k < 4)
4070         header[k++] = stbi__zget8(a);
4071     len = header[1] * 256 + header[0];
4072     nlen = header[3] * 256 + header[2];
4073     if (nlen != (len ^ 0xffff)) return stbi__err("zlib corrupt", "Corrupt PNG");
4074     if (a->zbuffer + len > a->zbuffer_end) return stbi__err("read past buffer", "Corrupt PNG");
4075     if (a->zout + len > a->zout_end)
4076         if (!stbi__zexpand(a, a->zout, len)) return 0;
4077     memcpy(a->zout, a->zbuffer, len);
4078     a->zbuffer += len;
4079     a->zout += len;
4080     return 1;
4081 }
4082
4083 static int stbi__parse_zlib_header(stbi__zbuf *a)
4084 {
4085     int cmf = stbi__zget8(a);
4086     int cm = cmf & 15;
4087     /* int cinfo = cmf >> 4; */
4088     int flg = stbi__zget8(a);
4089     if ((cmf * 256 + flg) % 31 != 0) return stbi__err("bad zlib header", "Corrupt PNG"); // zlib spec
4090     if (flg & 32) return stbi__err("no preset dict", "Corrupt PNG"); // preset dictionary not allowed
4091     in png
4092     if (cm != 8) return stbi__err("bad compression", "Corrupt PNG"); // DEFLATE required for png
4093                                     // window = 1 << (8 + cinfo)... but
4094     who cares, we fully buffer output
4095     return 1;
4096 }
4097
4098 // @TODO: should statically initialize these for optimal thread safety
4099 static stbi_uc stbi__zdefault_length[288], stbi__zdefault_distance[32];
4100 static void stbi__init_zdefaults(void)
4101 {
4102     int i; // use <= to match clearly with spec
4103     for (i = 0; i <= 143; ++i) stbi__zdefault_length[i] = 8;
4104     for (; i <= 255; ++i) stbi__zdefault_length[i] = 9;
4105     for (; i <= 279; ++i) stbi__zdefault_length[i] = 7;
4106     for (; i <= 287; ++i) stbi__zdefault_length[i] = 8;
4107
4108     for (i = 0; i <= 31; ++i) stbi__zdefault_distance[i] = 5;
4109 }
4110
4111 static int stbi__parse_zlib(stbi__zbuf *a, int parse_header)
4112 {
4113     int final, type;
4114     if (parse_header)
4115         if (!stbi__parse_zlib_header(a)) return 0;
4116     a->num_bits = 0;
4117     a->code_buffer = 0;
4118     do {
4119         final = stbi__zreceive(a, 1);
4120         type = stbi__zreceive(a, 2);
4121         if (type == 0) {
4122             if (!stbi__parse_uncompressed_block(a)) return 0;
4123         }
4124         else if (type == 3) {
4125             return 0;
4126         }
4127         else {
4128             if (type == 1) {
4129                 // use fixed code lengths
4130                 if (!stbi__zbuild_huffman(&a->z_length, stbi__zdefault_length, 288)) return 0;
4131                 if (!stbi__zbuild_huffman(&a->z_distance, stbi__zdefault_distance, 32)) return 0;
4132             }
4133             else {
4134                 if (!stbi__compute_huffman_codes(a)) return 0;
4135             }
4136             if (!stbi__parse_huffman_block(a)) return 0;
4137         }
4138     } while (!final);
4139     return 1;
4140 }
4141
4142 static int stbi__do_zlib(stbi__zbuf *a, char *obuf, int olen, int exp, int parse_header)

```

```

4142 {
4143     a->zout_start = obuf;
4144     a->zout = obuf;
4145     a->zout_end = obuf + olen;
4146     a->z_expandable = exp;
4147
4148     return stbi__parse_zlib(a, parse_header);
4149 }
4150
4151 STBIDEF char *stbi_zlib_decode_malloc_guesssize(const char *buffer, int len, int initial_size, int
*outlen)
4152 {
4153     stbi__zbuf a;
4154     char *p = (char *)stbi__malloc(initial_size);
4155     if (p == NULL) return NULL;
4156     a.zbuffer = (stbi_uc *)buffer;
4157     a.zbuffer_end = (stbi_uc *)buffer + len;
4158     if (stbi__do_zlib(&a, p, initial_size, 1, 1)) {
4159         if (outlen) *outlen = (int)(a.zout - a.zout_start);
4160         return a.zout_start;
4161     }
4162     else {
4163         STBI_FREE(a.zout_start);
4164         return NULL;
4165     }
4166 }
4167
4168 STBIDEF char *stbi_zlib_decode_malloc(char const *buffer, int len, int *outlen)
4169 {
4170     return stbi_zlib_decode_malloc_guesssize(buffer, len, 16384, outlen);
4171 }
4172
4173 STBIDEF char *stbi_zlib_decode_malloc_guesssize_headerflag(const char *buffer, int len, int
initial_size, int *outlen, int parse_header)
4174 {
4175     stbi__zbuf a;
4176     char *p = (char *)stbi__malloc(initial_size);
4177     if (p == NULL) return NULL;
4178     a.zbuffer = (stbi_uc *)buffer;
4179     a.zbuffer_end = (stbi_uc *)buffer + len;
4180     if (stbi__do_zlib(&a, p, initial_size, 1, parse_header)) {
4181         if (outlen) *outlen = (int)(a.zout - a.zout_start);
4182         return a.zout_start;
4183     }
4184     else {
4185         STBI_FREE(a.zout_start);
4186         return NULL;
4187     }
4188 }
4189
4190 STBIDEF int stbi_zlib_decode_buffer(char *obuffer, int olen, char const *ibuffer, int ilen)
4191 {
4192     stbi__zbuf a;
4193     a.zbuffer = (stbi_uc *)ibuffer;
4194     a.zbuffer_end = (stbi_uc *)ibuffer + ilen;
4195     if (stbi__do_zlib(&a, obuffer, olen, 0, 1))
4196         return (int)(a.zout - a.zout_start);
4197     else
4198         return -1;
4199 }
4200
4201 STBIDEF char *stbi_zlib_decode_noheader_malloc(char const *buffer, int len, int *outlen)
4202 {
4203     stbi__zbuf a;
4204     char *p = (char *)stbi__malloc(16384);
4205     if (p == NULL) return NULL;
4206     a.zbuffer = (stbi_uc *)buffer;
4207     a.zbuffer_end = (stbi_uc *)buffer + len;
4208     if (stbi__do_zlib(&a, p, 16384, 1, 0)) {
4209         if (outlen) *outlen = (int)(a.zout - a.zout_start);
4210         return a.zout_start;
4211     }
4212     else {
4213         STBI_FREE(a.zout_start);
4214         return NULL;
4215     }
4216 }
4217
4218 STBIDEF int stbi_zlib_decode_noheader_buffer(char *obuffer, int olen, const char *ibuffer, int ilen)
4219 {
4220     stbi__zbuf a;
4221     a.zbuffer = (stbi_uc *)ibuffer;
4222     a.zbuffer_end = (stbi_uc *)ibuffer + ilen;
4223     if (stbi__do_zlib(&a, obuffer, olen, 0, 0))
4224         return (int)(a.zout - a.zout_start);
4225     else
4226         return -1;

```

```

4227 }
4228 #endif
4229
4230 // public domain "baseline" PNG decoder v0.10 Sean Barrett 2006-11-18
4231 // simple implementation
4232 // - only 8-bit samples
4233 // - no CRC checking
4234 // - allocates lots of intermediate memory
4235 // - avoids problem of streaming data between subsystems
4236 // - avoids explicit window management
4237 // performance
4238 // - uses stb_zlib, a PD zlib implementation with fast huffman decoding
4239
4240 #ifndef STBI_NO_PNG
4241 typedef struct
4242 {
4243     stbi_uint32 length;
4244     stbi_uint32 type;
4245 } stbi_pngchunk;
4246
4247 static stbi_pngchunk stbi_get_chunk_header(stbi_context *s)
4248 {
4249     stbi_pngchunk c;
4250     c.length = stbi_get32be(s);
4251     c.type = stbi_get32be(s);
4252     return c;
4253 }
4254
4255 static int stbi_check_png_header(stbi_context *s)
4256 {
4257     static stbi_uc png_sig[8] = { 137,80,78,71,13,10,26,10 };
4258     int i;
4259     for (i = 0; i < 8; ++i)
4260         if (stbi_get8(s) != png_sig[i]) return stbi_err("bad png sig", "Not a PNG");
4261     return 1;
4262 }
4263
4264 typedef struct
4265 {
4266     stbi_context *s;
4267     stbi_uc *idata, *expanded, *out;
4268     int depth;
4269 } stbi_png;
4270
4271
4272 enum {
4273     STBI_F_none = 0,
4274     STBI_F_sub = 1,
4275     STBI_F_up = 2,
4276     STBI_F_avg = 3,
4277     STBI_F_paeth = 4,
4278     // synthetic filters used for first scanline to avoid needing a dummy row of 0s
4279     STBI_F_avg_first,
4280     STBI_F_paeth_first
4281 };
4282
4283 static stbi_uc first_row_filter[5] =
4284 {
4285     STBI_F_none,
4286     STBI_F_sub,
4287     STBI_F_none,
4288     STBI_F_avg_first,
4289     STBI_F_paeth_first
4290 };
4291
4292 static int stbi_paeth(int a, int b, int c)
4293 {
4294     int p = a + b - c;
4295     int pa = abs(p - a);
4296     int pb = abs(p - b);
4297     int pc = abs(p - c);
4298     if (pa <= pb && pa <= pc) return a;
4299     if (pb <= pc) return b;
4300     return c;
4301 }
4302
4303 static stbi_uc stbi_depth_scale_table[9] = { 0, 0xff, 0x55, 0, 0x11, 0,0,0, 0x01 };
4304
4305 // create the png data from post-deflated data
4306 static int stbi_create_png_image_raw(stbi_png *a, stbi_uc *raw, stbi_uint32 raw_len, int out_n,
4307     stbi_uint32 x, stbi_uint32 y, int depth, int color)
4308 {
4309     int bytes = (depth == 16 ? 2 : 1);
4310     stbi_context *s = a->s;
4311     stbi_uint32 i, j, stride = x*out_n*bytes;
4312     stbi_uint32 img_len, img_width_bytes;
4313     int k;

```

```

4313     int img_n = s->img_n; // copy it into a local for later
4314
4315     int output_bytes = out_n*bytes;
4316     int filter_bytes = img_n*bytes;
4317     int width = x;
4318
4319     STBI_ASSERT(out_n == s->img_n || out_n == s->img_n + 1);
4320     a->out = (stbi_uc *)stbi__malloc_mad3(x, y, output_bytes, 0); // extra bytes to write off the end
4321     into
4322     if (!a->out) return stbi__err("outofmem", "Out of memory");
4323
4324     img_width_bytes = (((img_n * x * depth) + 7) >> 3);
4325     img_len = (img_width_bytes + 1) * y;
4326     if (s->img_x == x && s->img_y == y) {
4327         if (raw_len != img_len) return stbi__err("not enough pixels", "Corrupt PNG");
4328     }
4329     else { // interlaced:
4330         if (raw_len < img_len) return stbi__err("not enough pixels", "Corrupt PNG");
4331     }
4332
4333     for (j = 0; j < y; ++j) {
4334         stbi_uc *cur = a->out + stride*j;
4335         stbi_uc *prior = cur - stride;
4336         int filter = *raw++;
4337
4338         if (filter > 4)
4339             return stbi__err("invalid filter", "Corrupt PNG");
4340
4341         if (depth < 8) {
4342             STBI_ASSERT(img_width_bytes <= x);
4343             cur += x*out_n - img_width_bytes; // store output to the rightmost img_len bytes, so we can
4344             decode in place
4345             filter_bytes = 1;
4346             width = img_width_bytes;
4347         }
4348
4349         // if first row, use special filter that doesn't sample previous row
4350         if (j == 0) filter = first_row_filter[filter];
4351
4352         // handle first byte explicitly
4353         for (k = 0; k < filter_bytes; ++k) {
4354             switch (filter) {
4355                 case STBI_F_none: cur[k] = raw[k]; break;
4356                 case STBI_F_sub: cur[k] = raw[k]; break;
4357                 case STBI_F_up: cur[k] = STBI_BYTECAST(raw[k] + prior[k]); break;
4358                 case STBI_F_avg: cur[k] = STBI_BYTECAST(raw[k] + (prior[k] >> 1)); break;
4359                 case STBI_F_paeth: cur[k] = STBI_BYTECAST(raw[k] + stbi_paeth(0, prior[k], 0)); break;
4360                 case STBI_F_avg_first: cur[k] = raw[k]; break;
4361                 case STBI_F_paeth_first: cur[k] = raw[k]; break;
4362             }
4363         }
4364
4365         if (depth == 8) {
4366             if (img_n != out_n)
4367                 cur[img_n] = 255; // first pixel
4368             raw += img_n;
4369             cur += out_n;
4370             prior += out_n;
4371         }
4372         else if (depth == 16) {
4373             if (img_n != out_n) {
4374                 cur[filter_bytes] = 255; // first pixel top byte
4375                 cur[filter_bytes + 1] = 255; // first pixel bottom byte
4376             }
4377             raw += filter_bytes;
4378             cur += output_bytes;
4379             prior += output_bytes;
4380         }
4381         else {
4382             raw += 1;
4383             cur += 1;
4384             prior += 1;
4385         }
4386
4387         // this is a little gross, so that we don't switch per-pixel or per-component
4388         if (depth < 8 || img_n == out_n) {
4389             int nk = (width - 1)*filter_bytes;
4390             #define STBI__CASE(f) \
4391             case f: \
4392                 for (k=0; k < nk; ++k) \
4393                     switch (filter) { \
4394                         // "none" filter turns into a memcpy here; make that explicit. \
4395                         case STBI_F_none: memcpy(cur, raw, nk); break; \
4396                         STBI__CASE(STBI_F_sub) { cur[k] = STBI_BYTECAST(raw[k] + cur[k - filter_bytes]); } \
4397                         STBI__CASE(STBI_F_up) { cur[k] = STBI_BYTECAST(raw[k] + prior[k]); } break; \
4398                         STBI__CASE(STBI_F_avg) { cur[k] = STBI_BYTECAST(raw[k] + ((prior[k] + cur[k -

```

```

    filter_bytes]) >> 1)); } break;
4397     STBI__CASE(STBI__F_paeth) { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k -
filter_bytes], prior[k], prior[k - filter_bytes])); } break;
4398     STBI__CASE(STBI__F_avg_first) { cur[k] = STBI__BYTECAST(raw[k] + (cur[k - filter_bytes]
>> 1)); } break;
4399     STBI__CASE(STBI__F_paeth_first) { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k -
filter_bytes], 0, 0)); } break;
4400 }
4401 #undef STBI__CASE
4402     raw += nk;
4403 }
4404     else {
4405         STBI_ASSERT(img_n + 1 == out_n);
4406 #define STBI__CASE(f) \
4407         case f: \
4408             for (i=x-1; i >= 1; --i,
cur[filter_bytes]=255,raw+=filter_bytes,cur+=output_bytes,prior+=output_bytes) \
4409                 for (k=0; k < filter_bytes; ++k)
4410                 switch (filter) {
4411                     STBI__CASE(STBI__F_none) { cur[k] = raw[k]; } break;
4412                     STBI__CASE(STBI__F_sub) { cur[k] = STBI__BYTECAST(raw[k] + cur[k - output_bytes]); }
break;
4413                     STBI__CASE(STBI__F_up) { cur[k] = STBI__BYTECAST(raw[k] + prior[k]); } break;
4414                     STBI__CASE(STBI__F_avg) { cur[k] = STBI__BYTECAST(raw[k] + ((prior[k] + cur[k -
output_bytes]) >> 1)); } break;
4415                     STBI__CASE(STBI__F_paeth) { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k -
output_bytes], prior[k], prior[k - output_bytes])); } break;
4416                     STBI__CASE(STBI__F_avg_first) { cur[k] = STBI__BYTECAST(raw[k] + (cur[k - output_bytes]
>> 1)); } break;
4417                     STBI__CASE(STBI__F_paeth_first) { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k -
output_bytes], 0, 0)); } break;
4418                 }
4419 #undef STBI__CASE
4420
4421         // the loop above sets the high byte of the pixels' alpha, but for
4422         // 16 bit png files we also need the low byte set. we'll do that here.
4423         if (depth == 16) {
4424             cur = a->out + stride*j; // start at the beginning of the row again
4425             for (i = 0; i < x; ++i, cur += output_bytes) {
4426                 cur[filter_bytes + 1] = 255;
4427             }
4428         }
4429     }
4430 }
4431
4432 // we make a separate pass to expand bits to pixels; for performance,
4433 // this could run two scanlines behind the above code, so it won't
4434 // interfere with filtering but will still be in the cache.
4435 if (depth < 8) {
4436     for (j = 0; j < y; ++j) {
4437         stbi_uc *cur = a->out + stride*j;
4438         stbi_uc *in = a->out + stride*j + x*out_n - img_width_bytes;
4439         // 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
4440         // 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
4441         stbi_uc scale = (color == 0) ? stbi__depth_scale_table[depth] : 1; // scale grayscale
values to 0..255 range
4442
4443                                     // note that the final
byte might overshoot and write more data than desired.
4444                                     // we can allocate
enough data that this never writes out of memory, but it
4445                                     // could also overwrite
the next scanline. can it overwrite non-empty data
4446                                     // on the next scanline?
yes, consider 1-pixel-wide scanlines with 1-bit-per-pixel.
4447                                     // so we need to
explicitly clamp the final ones
4448
4449         if (depth == 4) {
4450             for (k = x*img_n; k >= 2; k -= 2, ++in) {
4451                 *cur++ = scale * ((*in >> 4));
4452                 *cur++ = scale * ((*in & 0xf));
4453             }
4454             if (k > 0) *cur++ = scale * ((*in >> 4));
4455         }
4456         else if (depth == 2) {
4457             for (k = x*img_n; k >= 4; k -= 4, ++in) {
4458                 *cur++ = scale * ((*in >> 6));
4459                 *cur++ = scale * ((*in >> 4) & 0x03);
4460                 *cur++ = scale * ((*in >> 2) & 0x03);
4461                 *cur++ = scale * ((*in & 0x03));
4462             }
4463             if (k > 0) *cur++ = scale * ((*in >> 6));
4464             if (k > 1) *cur++ = scale * ((*in >> 4) & 0x03);
4465             if (k > 2) *cur++ = scale * ((*in >> 2) & 0x03);

```



```

4466     }
4467     else if (depth == 1) {
4468         for (k = x*img_n; k >= 8; k -= 8, ++in) {
4469             *cur++ = scale * ((*in >> 7));
4470             *cur++ = scale * ((*in >> 6) & 0x01);
4471             *cur++ = scale * ((*in >> 5) & 0x01);
4472             *cur++ = scale * ((*in >> 4) & 0x01);
4473             *cur++ = scale * ((*in >> 3) & 0x01);
4474             *cur++ = scale * ((*in >> 2) & 0x01);
4475             *cur++ = scale * ((*in >> 1) & 0x01);
4476             *cur++ = scale * ((*in) & 0x01);
4477         }
4478         if (k > 0) *cur++ = scale * ((*in >> 7));
4479         if (k > 1) *cur++ = scale * ((*in >> 6) & 0x01);
4480         if (k > 2) *cur++ = scale * ((*in >> 5) & 0x01);
4481         if (k > 3) *cur++ = scale * ((*in >> 4) & 0x01);
4482         if (k > 4) *cur++ = scale * ((*in >> 3) & 0x01);
4483         if (k > 5) *cur++ = scale * ((*in >> 2) & 0x01);
4484         if (k > 6) *cur++ = scale * ((*in >> 1) & 0x01);
4485     }
4486     if (img_n != out_n) {
4487         int q;
4488         // insert alpha = 255
4489         cur = a->out + stride*j;
4490         if (img_n == 1) {
4491             for (q = x - 1; q >= 0; --q) {
4492                 cur[q * 2 + 1] = 255;
4493                 cur[q * 2 + 0] = cur[q];
4494             }
4495         }
4496         else {
4497             STBI_ASSERT(img_n == 3);
4498             for (q = x - 1; q >= 0; --q) {
4499                 cur[q * 4 + 3] = 255;
4500                 cur[q * 4 + 2] = cur[q * 3 + 2];
4501                 cur[q * 4 + 1] = cur[q * 3 + 1];
4502                 cur[q * 4 + 0] = cur[q * 3 + 0];
4503             }
4504         }
4505     }
4506 }
4507 }
4508 else if (depth == 16) {
4509     // force the image data from big-endian to platform-native.
4510     // this is done in a separate pass due to the decoding relying
4511     // on the data being untouched, but could probably be done
4512     // per-line during decode if care is taken.
4513     stbi_uc *cur = a->out;
4514     stbi_uint16 *cur16 = (stbi_uint16*)cur;
4515
4516     for (i = 0; i < x*y*out_n; ++i, cur16++, cur += 2) {
4517         *cur16 = (cur[0] << 8) | cur[1];
4518     }
4519 }
4520
4521 return 1;
4522 }
4523
4524 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)
4525 {
4526     int bytes = (depth == 16 ? 2 : 1);
4527     int out_bytes = out_n * bytes;
4528     stbi_uc *final;
4529     int p;
4530     if (!interlaced)
4531         return stbi__create_png_image_raw(a, image_data, image_data_len, out_n, a->s->img_x,
            a->s->img_y, depth, color);
4532
4533     // de-interlacing
4534     final = (stbi_uc *)stbi__malloc_mad3(a->s->img_x, a->s->img_y, out_bytes, 0);
4535     for (p = 0; p < 7; ++p) {
4536         int xorig[] = { 0,4,0,2,0,1,0 };
4537         int yorig[] = { 0,0,4,0,2,0,1 };
4538         int xspc[] = { 8,8,4,4,2,2,1 };
4539         int yspc[] = { 8,8,8,4,4,2,2 };
4540         int i, j, x, y;
4541         // pass1_x[4] = 0, pass1_x[5] = 1, pass1_x[12] = 1
4542         x = (a->s->img_x - xorig[p] + xspc[p] - 1) / xspc[p];
4543         y = (a->s->img_y - yorig[p] + yspc[p] - 1) / yspc[p];
4544         if (x && y) {
4545             stbi_uint32 img_len = (((a->s->img_n * x * depth) + 7) >> 3) + 1 * y;
4546             if (!stbi__create_png_image_raw(a, image_data, image_data_len, out_n, x, y, depth, color))
4547                 return 0;
4548             STBI_FREE(final);
4549             return 0;
4550         }
4551     }
4552     return 0;
4553 }

```

```

4550         for (j = 0; j < y; ++j) {
4551             for (i = 0; i < x; ++i) {
4552                 int out_y = j*yspc[p] + yorig[p];
4553                 int out_x = i*xspc[p] + xorig[p];
4554                 memcpy(final + out_y*a->s->img_x*out_bytes + out_x*out_bytes,
4555                     a->out + (j*x + i)*out_bytes, out_bytes);
4556             }
4557         }
4558         STBI_FREE(a->out);
4559         image_data += img_len;
4560         image_data_len -= img_len;
4561     }
4562 }
4563 a->out = final;
4564
4565 return 1;
4566 }
4567
4568 static int stbi__compute_transparency(stbi_png *z, stbi_uc tc[3], int out_n)
4569 {
4570     stbi__context *s = z->s;
4571     stbi_uint32 i, pixel_count = s->img_x * s->img_y;
4572     stbi_uc *p = z->out;
4573
4574     // compute color-based transparency, assuming we've
4575     // already got 255 as the alpha value in the output
4576     STBI_ASSERT(out_n == 2 || out_n == 4);
4577
4578     if (out_n == 2) {
4579         for (i = 0; i < pixel_count; ++i) {
4580             p[1] = (p[0] == tc[0] ? 0 : 255);
4581             p += 2;
4582         }
4583     }
4584     else {
4585         for (i = 0; i < pixel_count; ++i) {
4586             if (p[0] == tc[0] && p[1] == tc[1] && p[2] == tc[2])
4587                 p[3] = 0;
4588             p += 4;
4589         }
4590     }
4591     return 1;
4592 }
4593
4594 static int stbi__compute_transparency16(stbi_png *z, stbi_uint16 tc[3], int out_n)
4595 {
4596     stbi__context *s = z->s;
4597     stbi_uint32 i, pixel_count = s->img_x * s->img_y;
4598     stbi_uint16 *p = (stbi_uint16*)z->out;
4599
4600     // compute color-based transparency, assuming we've
4601     // already got 65535 as the alpha value in the output
4602     STBI_ASSERT(out_n == 2 || out_n == 4);
4603
4604     if (out_n == 2) {
4605         for (i = 0; i < pixel_count; ++i) {
4606             p[1] = (p[0] == tc[0] ? 0 : 65535);
4607             p += 2;
4608         }
4609     }
4610     else {
4611         for (i = 0; i < pixel_count; ++i) {
4612             if (p[0] == tc[0] && p[1] == tc[1] && p[2] == tc[2])
4613                 p[3] = 0;
4614             p += 4;
4615         }
4616     }
4617     return 1;
4618 }
4619
4620 static int stbi__expand_png_palette(stbi_png *a, stbi_uc *palette, int len, int pal_img_n)
4621 {
4622     stbi_uint32 i, pixel_count = a->s->img_x * a->s->img_y;
4623     stbi_uc *p, *temp_out, *orig = a->out;
4624
4625     p = (stbi_uc *)stbi__malloc_mad2(pixel_count, pal_img_n, 0);
4626     if (p == NULL) return stbi__err("outofmem", "Out of memory");
4627
4628     // between here and free(out) below, exiting would leak
4629     temp_out = p;
4630
4631     if (pal_img_n == 3) {
4632         for (i = 0; i < pixel_count; ++i) {
4633             int n = orig[i] * 4;
4634             p[0] = palette[n];
4635             p[1] = palette[n + 1];
4636             p[2] = palette[n + 2];

```

```

4637         p += 3;
4638     }
4639 }
4640 else {
4641     for (i = 0; i < pixel_count; ++i) {
4642         int n = orig[i] * 4;
4643         p[0] = palette[n];
4644         p[1] = palette[n + 1];
4645         p[2] = palette[n + 2];
4646         p[3] = palette[n + 3];
4647         p += 4;
4648     }
4649 }
4650 STBI_FREE(a->out);
4651 a->out = temp_out;
4652
4653 STBI_NOTUSED(len);
4654
4655 return 1;
4656 }
4657
4658 static int stbi_unpremultiply_on_load = 0;
4659 static int stbi_de_iphone_flag = 0;
4660
4661 STBIDEF void stbi_set_unpremultiply_on_load(int flag_true_if_should_unpremultiply)
4662 {
4663     stbi_unpremultiply_on_load = flag_true_if_should_unpremultiply;
4664 }
4665
4666 STBIDEF void stbi_convert_iphone_png_to_rgb(int flag_true_if_should_convert)
4667 {
4668     stbi_de_iphone_flag = flag_true_if_should_convert;
4669 }
4670
4671 static void stbi_de_iphone(stbi_png *z)
4672 {
4673     stbi_context *s = z->s;
4674     stbi_uint32 i, pixel_count = s->img_x * s->img_y;
4675     stbi_uc *p = z->out;
4676
4677     if (s->img_out_n == 3) { // convert bgr to rgb
4678         for (i = 0; i < pixel_count; ++i) {
4679             stbi_uc t = p[0];
4680             p[0] = p[2];
4681             p[2] = t;
4682             p += 3;
4683         }
4684     }
4685     else {
4686         STBI_ASSERT(s->img_out_n == 4);
4687         if (stbi_unpremultiply_on_load) {
4688             // convert bgr to rgb and unpremultiply
4689             for (i = 0; i < pixel_count; ++i) {
4690                 stbi_uc a = p[3];
4691                 stbi_uc t = p[0];
4692                 if (a) {
4693                     p[0] = p[2] * 255 / a;
4694                     p[1] = p[1] * 255 / a;
4695                     p[2] = t * 255 / a;
4696                 }
4697                 else {
4698                     p[0] = p[2];
4699                     p[2] = t;
4700                 }
4701                 p += 4;
4702             }
4703         }
4704         else {
4705             // convert bgr to rgb
4706             for (i = 0; i < pixel_count; ++i) {
4707                 stbi_uc t = p[0];
4708                 p[0] = p[2];
4709                 p[2] = t;
4710                 p += 4;
4711             }
4712         }
4713     }
4714 }
4715
4716 #define STBI__PNG_TYPE(a,b,c,d) (((a) << 24) + ((b) << 16) + ((c) << 8) + (d))
4717
4718 static int stbi_parse_png_file(stbi_png *z, int scan, int req_comp)
4719 {
4720     stbi_uc palette[1024], pal_img_n = 0;
4721     stbi_uc has_trans = 0, tc[3];
4722     stbi_uint16 tc16[3];
4723     stbi_uint32 ioff = 0, idata_limit = 0, i, pal_len = 0;

```

```

4724     int first = 1, k, interlace = 0, color = 0, is_iphone = 0;
4725     stbi__context *s = z->s;
4726
4727     z->expanded = NULL;
4728     z->idata = NULL;
4729     z->out = NULL;
4730
4731     if (!stbi__check_png_header(s)) return 0;
4732
4733     if (scan == STBI__SCAN_type) return 1;
4734
4735     for (;;) {
4736         stbi__pngchunk c = stbi__get_chunk_header(s);
4737         switch (c.type) {
4738             case STBI__PNG_TYPE('C', 'g', 'B', 'I'):
4739                 is_iphone = 1;
4740                 stbi__skip(s, c.length);
4741                 break;
4742             case STBI__PNG_TYPE('I', 'H', 'D', 'R'): {
4743                 int comp, filter;
4744                 if (!first) return stbi__err("multiple IHDR", "Corrupt PNG");
4745                 first = 0;
4746                 if (c.length != 13) return stbi__err("bad IHDR len", "Corrupt PNG");
4747                 s->img_x = stbi__get32be(s); if (s->img_x > (1 << 24)) return stbi__err("too large", "Very
large image (corrupt?)");
4748                 s->img_y = stbi__get32be(s); if (s->img_y > (1 << 24)) return stbi__err("too large", "Very
large image (corrupt?)");
4749                 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");
4750                 color = stbi__get8(s); if (color > 6) return stbi__err("bad ctype", "Corrupt
PNG");
4751                 if (color == 3 && z->depth == 16) return stbi__err("bad ctype", "Corrupt
PNG");
4752                 if (color == 3) pal_img_n = 3; else if (color & 1) return stbi__err("bad ctype", "Corrupt
PNG");
4753                 comp = stbi__get8(s); if (comp) return stbi__err("bad comp method", "Corrupt PNG");
4754                 filter = stbi__get8(s); if (filter) return stbi__err("bad filter method", "Corrupt PNG");
4755                 interlace = stbi__get8(s); if (interlace > 1) return stbi__err("bad interlace method",
"Corrupt PNG");
4756                 if (!s->img_x || !s->img_y) return stbi__err("0-pixel image", "Corrupt PNG");
4757                 if (!pal_img_n) {
4758                     s->img_n = (color & 2 ? 3 : 1) + (color & 4 ? 1 : 0);
4759                     if ((1 << 30) / s->img_x / s->img_n < s->img_y) return stbi__err("too large", "Image too
large to decode");
4760                     if (scan == STBI__SCAN_header) return 1;
4761                 }
4762                 else {
4763                     // if paletted, then pal_n is our final components, and
4764                     // img_n is # components to decompress/filter.
4765                     s->img_n = 1;
4766                     if ((1 << 30) / s->img_x / 4 < s->img_y) return stbi__err("too large", "Corrupt PNG");
4767                     // if SCAN_header, have to scan to see if we have a tRNS
4768                 }
4769                 break;
4770             }
4771
4772             case STBI__PNG_TYPE('P', 'L', 'T', 'E'): {
4773                 if (first) return stbi__err("first not IHDR", "Corrupt PNG");
4774                 if (c.length > 256 * 3) return stbi__err("invalid PLTE", "Corrupt PNG");
4775                 pal_len = c.length / 3;
4776                 if (pal_len * 3 != c.length) return stbi__err("invalid PLTE", "Corrupt PNG");
4777                 for (i = 0; i < pal_len; ++i) {
4778                     palette[i * 4 + 0] = stbi__get8(s);
4779                     palette[i * 4 + 1] = stbi__get8(s);
4780                     palette[i * 4 + 2] = stbi__get8(s);
4781                     palette[i * 4 + 3] = 255;
4782                 }
4783                 break;
4784             }
4785
4786             case STBI__PNG_TYPE('t', 'R', 'N', 'S'): {
4787                 if (first) return stbi__err("first not IHDR", "Corrupt PNG");
4788                 if (z->idata) return stbi__err("tRNS after IDAT", "Corrupt PNG");
4789                 if (pal_img_n) {
4790                     if (scan == STBI__SCAN_header) { s->img_n = 4; return 1; }
4791                     if (pal_len == 0) return stbi__err("tRNS before PLTE", "Corrupt PNG");
4792                     if (c.length > pal_len) return stbi__err("bad tRNS len", "Corrupt PNG");
4793                     pal_img_n = 4;
4794                     for (i = 0; i < c.length; ++i)
4795                         palette[i * 4 + 3] = stbi__get8(s);
4796                 }
4797                 else {
4798                     if (!(s->img_n & 1)) return stbi__err("tRNS with alpha", "Corrupt PNG");
4799                     if (c.length != (stbi__uint32)s->img_n * 2) return stbi__err("bad tRNS len", "Corrupt
PNG");
4800                     has_trans = 1;

```

```

4801         if (z->depth == 16) {
4802             for (k = 0; k < s->img_n; ++k) tc16[k] = (stbi_uint16)stbi__get16be(s); // copy
the values as-is
4803         }
4804         else {
4805             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
4806         }
4807     }
4808     break;
4809 }
4810
4811     case STBI_PNG_TYPE('I', 'D', 'A', 'T'): {
4812         if (first) return stbi__err("first not IHDR", "Corrupt PNG");
4813         if (pal_img_n && !pal_len) return stbi__err("no PLTE", "Corrupt PNG");
4814         if (scan == STBI_SCAN_header) { s->img_n = pal_img_n; return 1; }
4815         if ((int)(ioff + c.length) < (int)ioff) return 0;
4816         if (ioff + c.length > idata_limit) {
4817             stbi_uint32 idata_limit_old = idata_limit;
4818             stbi_uc *p;
4819             if (idata_limit == 0) idata_limit = c.length > 4096 ? c.length : 4096;
4820             while (ioff + c.length > idata_limit)
4821                 idata_limit *= 2;
4822             STBI_NOTUSED(idata_limit_old);
4823             p = (stbi_uc *)STBI_REALLOC_SIZED(z->idata, idata_limit_old, idata_limit); if (p ==
NULL) return stbi__err("outofmem", "Out of memory");
4824             z->idata = p;
4825         }
4826         if (!stbi__getn(s, z->idata + ioff, c.length)) return stbi__err("outofdata", "Corrupt
PNG");
4827         ioff += c.length;
4828         break;
4829     }
4830
4831     case STBI_PNG_TYPE('I', 'E', 'N', 'D'): {
4832         stbi_uint32 raw_len, bpl;
4833         if (first) return stbi__err("first not IHDR", "Corrupt PNG");
4834         if (scan != STBI_SCAN_load) return 1;
4835         if (z->idata == NULL) return stbi__err("no IDAT", "Corrupt PNG");
4836         // initial guess for decoded data size to avoid unnecessary reallocs
4837         bpl = (s->img_x * z->depth + 7) / 8; // bytes per line, per component
4838         raw_len = bpl * s->img_y * s->img_n /* pixels */ + s->img_y /* filter mode per row */;
4839         z->expanded = (stbi_uc *)stbi_zlib_decode_malloc_guesssize_headerflag((char *)z->idata,
ioff, raw_len, (int *)&raw_len, !is_iphone);
4840         if (z->expanded == NULL) return 0; // zlib should set error
4841         STBI_FREE(z->idata); z->idata = NULL;
4842         if ((req_comp == s->img_n + 1 && req_comp != 3 && !pal_img_n) || has_trans)
4843             s->img_out_n = s->img_n + 1;
4844         else
4845             s->img_out_n = s->img_n;
4846         if (!stbi__create_png_image(z, z->expanded, raw_len, s->img_out_n, z->depth, color,
interlace)) return 0;
4847         if (has_trans) {
4848             if (z->depth == 16) {
4849                 if (!stbi__compute_transparency16(z, tc16, s->img_out_n)) return 0;
4850             }
4851             else {
4852                 if (!stbi__compute_transparency(z, tc, s->img_out_n)) return 0;
4853             }
4854         }
4855         if (is_iphone && stbi__de_iphone_flag && s->img_out_n > 2)
4856             stbi__de_iphone(z);
4857         if (pal_img_n) {
4858             // pal_img_n == 3 or 4
4859             s->img_n = pal_img_n; // record the actual colors we had
4860             s->img_out_n = pal_img_n;
4861             if (req_comp >= 3) s->img_out_n = req_comp;
4862             if (!stbi__expand_png_palette(z, palette, pal_len, s->img_out_n))
4863                 return 0;
4864         }
4865         STBI_FREE(z->expanded); z->expanded = NULL;
4866         return 1;
4867     }
4868
4869     default:
4870         // if critical, fail
4871         if (first) return stbi__err("first not IHDR", "Corrupt PNG");
4872         if ((c.type & (1 << 29)) == 0) {
4873 #ifndef STBI_NO_FAILURE_STRINGS
4874             // not threadsafe
4875             static char invalid_chunk[] = "XXXX PNG chunk not known";
4876             invalid_chunk[0] = STBI__BYTECAST(c.type >> 24);
4877             invalid_chunk[1] = STBI__BYTECAST(c.type >> 16);
4878             invalid_chunk[2] = STBI__BYTECAST(c.type >> 8);
4879             invalid_chunk[3] = STBI__BYTECAST(c.type >> 0);
4880 #endif
4881             return stbi__err(invalid_chunk, "PNG not supported: unknown PNG chunk type");

```

```

4882         }
4883         stbi__skip(s, c.length);
4884         break;
4885     }
4886     // end of PNG chunk, read and skip CRC
4887     stbi__get32be(s);
4888 }
4889 }
4890
4891 static void *stbi__do_png(stbi__png *p, int *x, int *y, int *n, int req_comp, stbi__result_info *ri)
4892 {
4893     void *result = NULL;
4894     if (req_comp < 0 || req_comp > 4) return stbi__errpuc("bad req_comp", "Internal error");
4895     if (stbi__parse_png_file(p, STBI__SCAN_load, req_comp)) {
4896         if (p->depth < 8)
4897             ri->bits_per_channel = 8;
4898         else
4899             ri->bits_per_channel = p->depth;
4900         result = p->out;
4901         p->out = NULL;
4902         if (req_comp && req_comp != p->s->img_out_n) {
4903             if (ri->bits_per_channel == 8)
4904                 result = stbi__convert_format((unsigned char *)result, p->s->img_out_n, req_comp,
4905                     p->s->img_x, p->s->img_y);
4906             else
4907                 result = stbi__convert_format16((stbi__uint16 *)result, p->s->img_out_n, req_comp,
4908                     p->s->img_x, p->s->img_y);
4909             p->s->img_out_n = req_comp;
4910             if (result == NULL) return result;
4911             *x = p->s->img_x;
4912             *y = p->s->img_y;
4913             if (n) *n = p->s->img_n;
4914         }
4915         STBI_FREE(p->out); p->out = NULL;
4916         STBI_FREE(p->expanded); p->expanded = NULL;
4917         STBI_FREE(p->idata); p->idata = NULL;
4918         return result;
4919     }
4920 }
4921 static void *stbi__png_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
4922     stbi__result_info *ri)
4923 {
4924     stbi__png p;
4925     p.s = s;
4926     return stbi__do_png(&p, x, y, comp, req_comp, ri);
4927 }
4928 static int stbi__png_test(stbi__context *s)
4929 {
4930     int r;
4931     r = stbi__check_png_header(s);
4932     stbi__rewind(s);
4933     return r;
4934 }
4935
4936 static int stbi__png_info_raw(stbi__png *p, int *x, int *y, int *comp)
4937 {
4938     if (!stbi__parse_png_file(p, STBI__SCAN_header, 0)) {
4939         stbi__rewind(p->s);
4940         return 0;
4941     }
4942     if (x) *x = p->s->img_x;
4943     if (y) *y = p->s->img_y;
4944     if (comp) *comp = p->s->img_n;
4945     return 1;
4946 }
4947
4948 static int stbi__png_info(stbi__context *s, int *x, int *y, int *comp)
4949 {
4950     stbi__png p;
4951     p.s = s;
4952     return stbi__png_info_raw(&p, x, y, comp);
4953 }
4954 #endif
4955
4956 // Microsoft/Windows BMP image
4957
4958 #ifndef STBI_NO_BMP
4959 static int stbi__bmp_test_raw(stbi__context *s)
4960 {
4961     int r;
4962     int sz;
4963     if (stbi__get8(s) != 'B') return 0;
4964     if (stbi__get8(s) != 'M') return 0;
4965     stbi__get32le(s); // discard filesize

```

```

4966     stbi__get16le(s); // discard reserved
4967     stbi__get16le(s); // discard reserved
4968     stbi__get32le(s); // discard data offset
4969     sz = stbi__get32le(s);
4970     r = (sz == 12 || sz == 40 || sz == 56 || sz == 108 || sz == 124);
4971     return r;
4972 }
4973
4974 static int stbi__bmp_test(stbi__context *s)
4975 {
4976     int r = stbi__bmp_test_raw(s);
4977     stbi__rewind(s);
4978     return r;
4979 }
4980
4981 // returns 0..31 for the highest set bit
4982 static int stbi__high_bit(unsigned int z)
4983 {
4984     int n = 0;
4985     if (z == 0) return -1;
4986     if (z >= 0x10000) n += 16, z >>= 16;
4987     if (z >= 0x00100) n += 8, z >>= 8;
4988     if (z >= 0x00010) n += 4, z >>= 4;
4989     if (z >= 0x00004) n += 2, z >>= 2;
4990     if (z >= 0x00002) n += 1, z >>= 1;
4991     return n;
4992 }
4993 }
4994
4995 static int stbi__bitcount(unsigned int a)
4996 {
4997     a = (a & 0x55555555) + ((a >> 1) & 0x55555555); // max 2
4998     a = (a & 0x33333333) + ((a >> 2) & 0x33333333); // max 4
4999     a = (a + (a >> 4)) & 0x0f0f0f0f; // max 8 per 4, now 8 bits
5000     a = (a + (a >> 8)); // max 16 per 8 bits
5001     a = (a + (a >> 16)); // max 32 per 8 bits
5002     return a & 0xff;
5003 }
5004
5005 static int stbi__shiftsigned(int v, int shift, int bits)
5006 {
5007     int result;
5008     int z = 0;
5009
5010     if (shift < 0) v <<= -shift;
5011     else v >>= shift;
5012     result = v;
5013
5014     z = bits;
5015     while (z < 8) {
5016         result += v >> z;
5017         z += bits;
5018     }
5019     return result;
5020 }
5021
5022 typedef struct
5023 {
5024     int bpp, offset, hsz;
5025     unsigned int mr, mg, mb, ma, all_a;
5026 } stbi__bmp_data;
5027
5028 static void *stbi__bmp_parse_header(stbi__context *s, stbi__bmp_data *info)
5029 {
5030     int hsz;
5031     if (stbi__get8(s) != 'B' || stbi__get8(s) != 'M') return stbi__errpuc("not BMP", "Corrupt BMP");
5032     stbi__get32le(s); // discard filesize
5033     stbi__get16le(s); // discard reserved
5034     stbi__get16le(s); // discard reserved
5035     info->offset = stbi__get32le(s);
5036     info->hsz = hsz = stbi__get32le(s);
5037     info->mr = info->mg = info->mb = info->ma = 0;
5038
5039     if (hsz != 12 && hsz != 40 && hsz != 56 && hsz != 108 && hsz != 124) return stbi__errpuc("unknown BMP", "BMP type not supported: unknown");
5040     if (hsz == 12) {
5041         s->img_x = stbi__get16le(s);
5042         s->img_y = stbi__get16le(s);
5043     }
5044     else {
5045         s->img_x = stbi__get32le(s);
5046         s->img_y = stbi__get32le(s);
5047     }
5048     if (stbi__get16le(s) != 1) return stbi__errpuc("bad BMP", "bad BMP");
5049     info->bpp = stbi__get16le(s);
5050     if (info->bpp == 1) return stbi__errpuc("monochrome", "BMP type not supported: 1-bit");
5051     if (hsz != 12) {

```

```

5052     int compress = stbi__get32le(s);
5053     if (compress == 1 || compress == 2) return stbi__errpuc("BMP RLE", "BMP type not supported:
RLE");
5054     stbi__get32le(s); // discard sizeof
5055     stbi__get32le(s); // discard hres
5056     stbi__get32le(s); // discard vres
5057     stbi__get32le(s); // discard colorsused
5058     stbi__get32le(s); // discard max important
5059     if (hsz == 40 || hsz == 56) {
5060         if (hsz == 56) {
5061             stbi__get32le(s);
5062             stbi__get32le(s);
5063             stbi__get32le(s);
5064             stbi__get32le(s);
5065         }
5066         if (info->bpp == 16 || info->bpp == 32) {
5067             if (compress == 0) {
5068                 if (info->bpp == 32) {
5069                     info->mr = 0xffu << 16;
5070                     info->mg = 0xffu << 8;
5071                     info->mb = 0xffu << 0;
5072                     info->ma = 0xffu << 24;
5073                     info->all_a = 0; // if all_a is 0 at end, then we loaded alpha channel but it
was all 0
5074                 }
5075                 else {
5076                     info->mr = 31u << 10;
5077                     info->mg = 31u << 5;
5078                     info->mb = 31u << 0;
5079                 }
5080             }
5081             else if (compress == 3) {
5082                 info->mr = stbi__get32le(s);
5083                 info->mg = stbi__get32le(s);
5084                 info->mb = stbi__get32le(s);
5085                 // not documented, but generated by photoshop and handled by mspaint
5086                 if (info->mr == info->mg && info->mg == info->mb) {
5087                     // !?!?!
5088                     return stbi__errpuc("bad BMP", "bad BMP");
5089                 }
5090             }
5091             else
5092                 return stbi__errpuc("bad BMP", "bad BMP");
5093         }
5094     }
5095     else {
5096         int i;
5097         if (hsz != 108 && hsz != 124)
5098             return stbi__errpuc("bad BMP", "bad BMP");
5099         info->mr = stbi__get32le(s);
5100         info->mg = stbi__get32le(s);
5101         info->mb = stbi__get32le(s);
5102         info->ma = stbi__get32le(s);
5103         stbi__get32le(s); // discard color space
5104         for (i = 0; i < 12; ++i)
5105             stbi__get32le(s); // discard color space parameters
5106         if (hsz == 124) {
5107             stbi__get32le(s); // discard rendering intent
5108             stbi__get32le(s); // discard offset of profile data
5109             stbi__get32le(s); // discard size of profile data
5110             stbi__get32le(s); // discard reserved
5111         }
5112     }
5113 }
5114 return (void *)1;
5115 }
5116
5117
5118 static void *stbi__bmp_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
stbi__result_info *ri)
5119 {
5120     stbi_uc *out;
5121     unsigned int mr = 0, mg = 0, mb = 0, ma = 0, all_a;
5122     stbi_uc pal[256][4];
5123     int psize = 0, i, j, width;
5124     int flip_vertically, pad, target;
5125     stbi__bmp_data info;
5126     STBI_NOTUSED(ri);
5127
5128     info.all_a = 255;
5129     if (stbi__bmp_parse_header(s, &info) == NULL)
5130         return NULL; // error code already set
5131
5132     flip_vertically = ((int)s->img_y) > 0;
5133     s->img_y = abs((int)s->img_y);
5134
5135     mr = info.mr;

```



```

5136     mg = info.mg;
5137     mb = info.mb;
5138     ma = info.ma;
5139     all_a = info.all_a;
5140
5141     if (info.hsz == 12) {
5142         if (info.bpp < 24)
5143             psize = (info.offset - 14 - 24) / 3;
5144     }
5145     else {
5146         if (info.bpp < 16)
5147             psize = (info.offset - 14 - info.hsz) » 2;
5148     }
5149
5150     s->img_n = ma ? 4 : 3;
5151     if (req_comp && req_comp >= 3) // we can directly decode 3 or 4
5152         target = req_comp;
5153     else
5154         target = s->img_n; // if they want monochrome, we'll post-convert
5155
5156         // sanity-check size
5157     if (!stbi__mad3sizes_valid(target, s->img_x, s->img_y, 0))
5158         return stbi__errpuc("too large", "Corrupt BMP");
5159
5160     out = (stbi_uc *)stbi__malloc_mad3(target, s->img_x, s->img_y, 0);
5161     if (!out) return stbi__errpuc("outofmem", "Out of memory");
5162     if (info.bpp < 16) {
5163         int z = 0;
5164         if (psize == 0 || psize > 256) { STBI_FREE(out); return stbi__errpuc("invalid", "Corrupt BMP"); }
5165
5166         for (i = 0; i < psize; ++i) {
5167             pal[i][2] = stbi__get8(s);
5168             pal[i][1] = stbi__get8(s);
5169             pal[i][0] = stbi__get8(s);
5170             if (info.hsz != 12) stbi__get8(s);
5171             pal[i][3] = 255;
5172         }
5173         stbi__skip(s, info.offset - 14 - info.hsz - psize * (info.hsz == 12 ? 3 : 4));
5174         if (info.bpp == 4) width = (s->img_x + 1) » 1;
5175         else if (info.bpp == 8) width = s->img_x;
5176         else { STBI_FREE(out); return stbi__errpuc("bad bpp", "Corrupt BMP"); }
5177         pad = (-width) & 3;
5178         for (j = 0; j < (int)s->img_y; ++j) {
5179             for (i = 0; i < (int)s->img_x; i += 2) {
5180                 int v = stbi__get8(s), v2 = 0;
5181                 if (info.bpp == 4) {
5182                     v2 = v & 15;
5183                     v »= 4;
5184                 }
5185                 out[z++] = pal[v][0];
5186                 out[z++] = pal[v][1];
5187                 out[z++] = pal[v][2];
5188                 if (target == 4) out[z++] = 255;
5189                 if (i + 1 == (int)s->img_x) break;
5190                 v = (info.bpp == 8) ? stbi__get8(s) : v2;
5191                 out[z++] = pal[v][0];
5192                 out[z++] = pal[v][1];
5193                 out[z++] = pal[v][2];
5194                 if (target == 4) out[z++] = 255;
5195             }
5196             stbi__skip(s, pad);
5197         }
5198     }
5199     else {
5200         int rshift = 0, gshift = 0, bshift = 0, ashift = 0, rcount = 0, gcount = 0, bcount = 0, acount
5201         = 0;
5202         int z = 0;
5203         int easy = 0;
5204         stbi__skip(s, info.offset - 14 - info.hsz);
5205         if (info.bpp == 24) width = 3 * s->img_x;
5206         else if (info.bpp == 16) width = 2 * s->img_x;
5207         else /* bpp = 32 and pad = 0 */ width = 0;
5208         pad = (-width) & 3;
5209         if (info.bpp == 24) {
5210             easy = 1;
5211         }
5212         else if (info.bpp == 32) {
5213             if (mb == 0xff && mg == 0xff00 && mr == 0x00ff0000 && ma == 0xff000000)
5214                 easy = 2;
5215         }
5216         if (!easy) {
5217             if (!mr || !mg || !mb) { STBI_FREE(out); return stbi__errpuc("bad masks", "Corrupt BMP"); }
5218             // right shift amt to put high bit in position #7
5219             rshift = stbi__high_bit(mr) - 7; rcount = stbi__bitcount(mr);
5220             gshift = stbi__high_bit(mg) - 7; gcount = stbi__bitcount(mg);
5221             bshift = stbi__high_bit(mb) - 7; bcount = stbi__bitcount(mb);
5222             ashift = stbi__high_bit(ma) - 7; acount = stbi__bitcount(ma);

```

```

5221     }
5222     for (j = 0; j < (int)s->img_y; ++j) {
5223         if (easy) {
5224             for (i = 0; i < (int)s->img_x; ++i) {
5225                 unsigned char a;
5226                 out[z + 2] = stbi__get8(s);
5227                 out[z + 1] = stbi__get8(s);
5228                 out[z + 0] = stbi__get8(s);
5229                 z += 3;
5230                 a = (easy == 2 ? stbi__get8(s) : 255);
5231                 all_a |= a;
5232                 if (target == 4) out[z++] = a;
5233             }
5234         }
5235         else {
5236             int bpp = info.bpp;
5237             for (i = 0; i < (int)s->img_x; ++i) {
5238                 stbi__uint32 v = (bpp == 16 ? (stbi__uint32)stbi__get16le(s) : stbi__get32le(s));
5239                 int a;
5240                 out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mr, rshift, rcount));
5241                 out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mg, gshift, gcount));
5242                 out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mb, bshift, bcount));
5243                 a = (ma ? stbi__shiftsigned(v & ma, ashift, account) : 255);
5244                 all_a |= a;
5245                 if (target == 4) out[z++] = STBI__BYTECAST(a);
5246             }
5247         }
5248         stbi__skip(s, pad);
5249     }
5250 }
5251
5252 // if alpha channel is all 0s, replace with all 255s
5253 if (target == 4 && all_a == 0)
5254     for (i = 4 * s->img_x*s->img_y - 1; i >= 0; i -= 4)
5255         out[i] = 255;
5256
5257 if (flip_vertically) {
5258     stbi_uc t;
5259     for (j = 0; j < (int)s->img_y >> 1; ++j) {
5260         stbi_uc *p1 = out + j * s->img_x*target;
5261         stbi_uc *p2 = out + (s->img_y - 1 - j) * s->img_x*target;
5262         for (i = 0; i < (int)s->img_x*target; ++i) {
5263             t = p1[i], p1[i] = p2[i], p2[i] = t;
5264         }
5265     }
5266 }
5267
5268 if (req_comp && req_comp != target) {
5269     out = stbi__convert_format(out, target, req_comp, s->img_x, s->img_y);
5270     if (out == NULL) return out; // stbi__convert_format frees input on failure
5271 }
5272
5273 *x = s->img_x;
5274 *y = s->img_y;
5275 if (comp) *comp = s->img_n;
5276 return out;
5277 }
5278 #endif
5279
5280 // Targa Truevision - TGA
5281 // by Jonathan Dummer
5282 #ifndef STBI_NO_TGA
5283 // returns STBI_rgb or whatever, 0 on error
5284 static int stbi__tga_get_comp(int bits_per_pixel, int is_grey, int* is_rgb16)
5285 {
5286     // only RGB or RGBA (incl. 16bit) or grey allowed
5287     if (is_rgb16) *is_rgb16 = 0;
5288     switch (bits_per_pixel) {
5289         case 8: return STBI_grey;
5290         case 16: if (is_grey) return STBI_grey_alpha;
5291                 // else: fall-through
5292         case 15: if (is_rgb16) *is_rgb16 = 1;
5293                 return STBI_rgb;
5294         case 24: // fall-through
5295         case 32: return bits_per_pixel / 8;
5296         default: return 0;
5297     }
5298 }
5299
5300 static int stbi__tga_info(stbi__context *s, int *x, int *y, int *comp)
5301 {
5302     int tga_w, tga_h, tga_comp, tga_image_type, tga_bits_per_pixel, tga_colormap_bpp;
5303     int sz, tga_colormap_type;
5304     stbi__get8(s); // discard Offset
5305     tga_colormap_type = stbi__get8(s); // colormap type
5306     if (tga_colormap_type > 1) {
5307         stbi__rewind(s);

```

```

5308     return 0;        // only RGB or indexed allowed
5309 }
5310 tga_image_type = stbi__get8(s); // image type
5311 if (tga_colormap_type == 1) { // colormapped (paletted) image
5312     if (tga_image_type != 1 && tga_image_type != 9) {
5313         stbi__rewind(s);
5314         return 0;
5315     }
5316     stbi__skip(s, 4); // skip index of first colormap entry and number of entries
5317     sz = stbi__get8(s); // check bits per palette color entry
5318     if ((sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32)) {
5319         stbi__rewind(s);
5320         return 0;
5321     }
5322     stbi__skip(s, 4); // skip image x and y origin
5323     tga_colormap_bpp = sz;
5324 }
5325 else { // "normal" image w/o colormap - only RGB or grey allowed, +/- RLE
5326     if ((tga_image_type != 2) && (tga_image_type != 3) && (tga_image_type != 10) && (tga_image_type
!= 11)) {
5327         stbi__rewind(s);
5328         return 0; // only RGB or grey allowed, +/- RLE
5329     }
5330     stbi__skip(s, 9); // skip colormap specification and image x/y origin
5331     tga_colormap_bpp = 0;
5332 }
5333 tga_w = stbi__get16le(s);
5334 if (tga_w < 1) {
5335     stbi__rewind(s);
5336     return 0; // test width
5337 }
5338 tga_h = stbi__get16le(s);
5339 if (tga_h < 1) {
5340     stbi__rewind(s);
5341     return 0; // test height
5342 }
5343 tga_bits_per_pixel = stbi__get8(s); // bits per pixel
5344 stbi__get8(s); // ignore alpha bits
5345 if (tga_colormap_bpp != 0) {
5346     if ((tga_bits_per_pixel != 8) && (tga_bits_per_pixel != 16)) {
5347         // when using a colormap, tga_bits_per_pixel is the size of the indexes
5348         // I don't think anything but 8 or 16bit indexes makes sense
5349         stbi__rewind(s);
5350         return 0;
5351     }
5352     tga_comp = stbi__tga_get_comp(tga_colormap_bpp, 0, NULL);
5353 }
5354 else {
5355     tga_comp = stbi__tga_get_comp(tga_bits_per_pixel, (tga_image_type == 3) || (tga_image_type ==
11), NULL);
5356 }
5357 if (!tga_comp) {
5358     stbi__rewind(s);
5359     return 0;
5360 }
5361 if (x) *x = tga_w;
5362 if (y) *y = tga_h;
5363 if (comp) *comp = tga_comp;
5364 return 1; // seems to have passed everything
5365 }
5366
5367 static int stbi__tga_test(stbi__context *s)
5368 {
5369     int res = 0;
5370     int sz, tga_color_type;
5371     stbi__get8(s); // discard Offset
5372     tga_color_type = stbi__get8(s); // color type
5373     if (tga_color_type > 1) goto errorEnd; // only RGB or indexed allowed
5374     sz = stbi__get8(s); // image type
5375     if (tga_color_type == 1) { // colormapped (paletted) image
5376         if (sz != 1 && sz != 9) goto errorEnd; // colortype 1 demands image type 1 or 9
5377         stbi__skip(s, 4); // skip index of first colormap entry and number of entries
5378         sz = stbi__get8(s); // check bits per palette color entry
5379         if ((sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32)) goto errorEnd;
5380         stbi__skip(s, 4); // skip image x and y origin
5381     }
5382     else { // "normal" image w/o colormap
5383         if ((sz != 2) && (sz != 3) && (sz != 10) && (sz != 11)) goto errorEnd; // only RGB or grey
allowed, +/- RLE
5384         stbi__skip(s, 9); // skip colormap specification and image x/y origin
5385     }
5386     if (stbi__get16le(s) < 1) goto errorEnd; // test width
5387     if (stbi__get16le(s) < 1) goto errorEnd; // test height
5388     sz = stbi__get8(s); // bits per pixel
5389     if ((tga_color_type == 1) && (sz != 8) && (sz != 16)) goto errorEnd; // for colormapped images, bpp
is size of an index
5390     if ((sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32)) goto errorEnd;

```

```

5391
5392     res = 1; // if we got this far, everything's good and we can return 1 instead of 0
5393
5394 errorEnd:
5395     stbi__rewind(s);
5396     return res;
5397 }
5398
5399 // read 16bit value and convert to 24bit RGB
5400 static void stbi__tga_read_rgb16(stbi__context *s, stbi_uc* out)
5401 {
5402     stbi__uint16 px = (stbi__uint16)stbi__get16le(s);
5403     stbi__uint16 fiveBitMask = 31;
5404     // we have 3 channels with 5bits each
5405     int r = (px >> 10) & fiveBitMask;
5406     int g = (px >> 5) & fiveBitMask;
5407     int b = px & fiveBitMask;
5408     // Note that this saves the data in RGB(A) order, so it doesn't need to be swapped later
5409     out[0] = (stbi_uc)((r * 255) / 31);
5410     out[1] = (stbi_uc)((g * 255) / 31);
5411     out[2] = (stbi_uc)((b * 255) / 31);
5412
5413     // some people claim that the most significant bit might be used for alpha
5414     // (possibly if an alpha-bit is set in the "image descriptor byte")
5415     // but that only made 16bit test images completely translucent..
5416     // so let's treat all 15 and 16bit TGAs as RGB with no alpha.
5417 }
5418
5419 static void *stbi__tga_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
5420     stbi__result_info *ri)
5421 {
5422     // read in the TGA header stuff
5423     int tga_offset = stbi__get8(s);
5424     int tga_indexed = stbi__get8(s);
5425     int tga_image_type = stbi__get8(s);
5426     int tga_is_RLE = 0;
5427     int tga_palette_start = stbi__get16le(s);
5428     int tga_palette_len = stbi__get16le(s);
5429     int tga_palette_bits = stbi__get8(s);
5430     int tga_x_origin = stbi__get16le(s);
5431     int tga_y_origin = stbi__get16le(s);
5432     int tga_width = stbi__get16le(s);
5433     int tga_height = stbi__get16le(s);
5434     int tga_bits_per_pixel = stbi__get8(s);
5435     int tga_comp, tga_rgb16 = 0;
5436     int tga_inverted = stbi__get8(s);
5437     // int tga_alpha_bits = tga_inverted & 15; // the 4 lowest bits - unused (useless?)
5438     // image data
5439     unsigned char *tga_data;
5440     unsigned char *tga_palette = NULL;
5441     int i, j;
5442     unsigned char raw_data[4] = { 0 };
5443     int RLE_count = 0;
5444     int RLE_repeating = 0;
5445     int read_next_pixel = 1;
5446     STBI_NOTUSED(ri);
5447
5448     // do a tiny bit of precessing
5449     if (tga_image_type >= 8)
5450     {
5451         tga_image_type -= 8;
5452         tga_is_RLE = 1;
5453     }
5454     tga_inverted = 1 - ((tga_inverted >> 5) & 1);
5455
5456     // If I'm paletted, then I'll use the number of bits from the palette
5457     if (tga_indexed) tga_comp = stbi__tga_get_comp(tga_palette_bits, 0, &tga_rgb16);
5458     else tga_comp = stbi__tga_get_comp(tga_bits_per_pixel, (tga_image_type == 3), &tga_rgb16);
5459
5460     if (!tga_comp) // shouldn't really happen, stbi__tga_test() should have ensured basic consistency
5461         return stbi__errpuc("bad format", "Can't find out TGA pixelformat");
5462
5463     // tga info
5464     *x = tga_width;
5465     *y = tga_height;
5466     if (comp) *comp = tga_comp;
5467
5468     if (!stbi__mad3sizes_valid(tga_width, tga_height, tga_comp, 0))
5469         return stbi__errpuc("too large", "Corrupt TGA");
5470
5471     tga_data = (unsigned char*)stbi__malloc_mad3(tga_width, tga_height, tga_comp, 0);
5472     if (!tga_data) return stbi__errpuc("outofmem", "Out of memory");
5473
5474     // skip to the data's starting position (offset usually = 0)
5475     stbi__skip(s, tga_offset);
5476
5477     if (!tga_indexed && !tga_is_RLE && !tga_rgb16) {

```

```

5477     for (i = 0; i < tga_height; ++i) {
5478         int row = tga_inverted ? tga_height - i - 1 : i;
5479         stbi_uc *tga_row = tga_data + row*tga_width*tga_comp;
5480         stbi__getn(s, tga_row, tga_width * tga_comp);
5481     }
5482 }
5483 else {
5484     // do I need to load a palette?
5485     if (tga_indexed)
5486     {
5487         // any data to skip? (offset usually = 0)
5488         stbi__skip(s, tga_palette_start);
5489         // load the palette
5490         tga_palette = (unsigned char*)stbi__malloc_mad2(tga_palette_len, tga_comp, 0);
5491         if (!tga_palette) {
5492             STBI_FREE(tga_data);
5493             return stbi__errpuc("outofmem", "Out of memory");
5494         }
5495         if (tga_rgb16) {
5496             stbi_uc *pal_entry = tga_palette;
5497             STBI_ASSERT(tga_comp == STBI_rgb);
5498             for (i = 0; i < tga_palette_len; ++i) {
5499                 stbi__tga_read_rgb16(s, pal_entry);
5500                 pal_entry += tga_comp;
5501             }
5502         }
5503         else if (!stbi__getn(s, tga_palette, tga_palette_len * tga_comp)) {
5504             STBI_FREE(tga_data);
5505             STBI_FREE(tga_palette);
5506             return stbi__errpuc("bad palette", "Corrupt TGA");
5507         }
5508     }
5509     // load the data
5510     for (i = 0; i < tga_width * tga_height; ++i)
5511     {
5512         // if I'm in RLE mode, do I need to get a RLE stbi__pngchunk?
5513         if (tga_is_RLE)
5514         {
5515             if (RLE_count == 0)
5516             {
5517                 // yep, get the next byte as a RLE command
5518                 int RLE_cmd = stbi__get8(s);
5519                 RLE_count = 1 + (RLE_cmd & 127);
5520                 RLE_repeating = RLE_cmd > 7;
5521                 read_next_pixel = 1;
5522             }
5523             else if (!RLE_repeating)
5524             {
5525                 read_next_pixel = 1;
5526             }
5527         }
5528         else
5529         {
5530             read_next_pixel = 1;
5531         }
5532         // OK, if I need to read a pixel, do it now
5533         if (read_next_pixel)
5534         {
5535             // load however much data we did have
5536             if (tga_indexed)
5537             {
5538                 // read in index, then perform the lookup
5539                 int pal_idx = (tga_bits_per_pixel == 8) ? stbi__get8(s) : stbi__get16le(s);
5540                 if (pal_idx >= tga_palette_len) {
5541                     // invalid index
5542                     pal_idx = 0;
5543                 }
5544                 pal_idx *= tga_comp;
5545                 for (j = 0; j < tga_comp; ++j) {
5546                     raw_data[j] = tga_palette[pal_idx + j];
5547                 }
5548             }
5549             else if (tga_rgb16) {
5550                 STBI_ASSERT(tga_comp == STBI_rgb);
5551                 stbi__tga_read_rgb16(s, raw_data);
5552             }
5553             else {
5554                 // read in the data raw
5555                 for (j = 0; j < tga_comp; ++j) {
5556                     raw_data[j] = stbi__get8(s);
5557                 }
5558             }
5559             // clear the reading flag for the next pixel
5560             read_next_pixel = 0;
5561         } // end of reading a pixel
5562     }
5563     // copy data

```

```

5564         for (j = 0; j < tga_comp; ++j)
5565             tga_data[i*tga_comp + j] = raw_data[j];
5566
5567         // in case we're in RLE mode, keep counting down
5568         --RLE_count;
5569     }
5570     // do I need to invert the image?
5571     if (tga_inverted)
5572     {
5573         for (j = 0; j * 2 < tga_height; ++j)
5574         {
5575             int index1 = j * tga_width * tga_comp;
5576             int index2 = (tga_height - 1 - j) * tga_width * tga_comp;
5577             for (i = tga_width * tga_comp; i > 0; --i)
5578             {
5579                 unsigned char temp = tga_data[index1];
5580                 tga_data[index1] = tga_data[index2];
5581                 tga_data[index2] = temp;
5582                 ++index1;
5583                 ++index2;
5584             }
5585         }
5586     }
5587     // clear my palette, if I had one
5588     if (tga_palette != NULL)
5589     {
5590         STBI_FREE(tga_palette);
5591     }
5592 }
5593
5594 // swap RGB - if the source data was RGB16, it already is in the right order
5595 if (tga_comp >= 3 && !tga_rgb16)
5596 {
5597     unsigned char* tga_pixel = tga_data;
5598     for (i = 0; i < tga_width * tga_height; ++i)
5599     {
5600         unsigned char temp = tga_pixel[0];
5601         tga_pixel[0] = tga_pixel[2];
5602         tga_pixel[2] = temp;
5603         tga_pixel += tga_comp;
5604     }
5605 }
5606
5607 // convert to target component count
5608 if (req_comp && req_comp != tga_comp)
5609     tga_data = stbi__convert_format(tga_data, tga_comp, req_comp, tga_width, tga_height);
5610
5611 // the things I do to get rid of an error message, and yet keep
5612 // Microsoft's C compilers happy... [8^(
5613 tga_palette_start = tga_palette_len = tga_palette_bits =
5614 tga_x_origin = tga_y_origin = 0;
5615 // OK, done
5616 return tga_data;
5617 }
5618 #endif
5619
5620 // *****
5621 // Photoshop PSD loader -- PD by Thatcher Ulrich, integration by Nicolas Schulz, tweaked by STB
5622
5623 #ifndef STBI_NO_PSD
5624 static int stbi_psd_test(stbi__context *s)
5625 {
5626     int r = (stbi__get32be(s) == 0x38425053);
5627     stbi__rewind(s);
5628     return r;
5629 }
5630
5631 static int stbi_psd_decode_rle(stbi__context *s, stbi_uc *p, int pixelCount)
5632 {
5633     int count, nleft, len;
5634
5635     count = 0;
5636     while ((nleft = pixelCount - count) > 0) {
5637         len = stbi__get8(s);
5638         if (len == 128) {
5639             // No-op.
5640         }
5641         else if (len < 128) {
5642             // Copy next len+1 bytes literally.
5643             len++;
5644             if (len > nleft) return 0; // corrupt data
5645             count += len;
5646             while (len) {
5647                 *p = stbi__get8(s);
5648                 p += 4;
5649                 len--;
5650             }

```

```

5651     }
5652     else if (len > 128) {
5653         stbi_uc val;
5654         // Next -len+1 bytes in the dest are replicated from next source byte.
5655         // (Interpret len as a negative 8-bit int.)
5656         len = 257 - len;
5657         if (len > nleft) return 0; // corrupt data
5658         val = stbi__get8(s);
5659         count += len;
5660         while (len) {
5661             *p = val;
5662             p += 4;
5663             len--;
5664         }
5665     }
5666 }
5667
5668 return 1;
5669 }
5670
5671 static void stbi__psd_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
    stbi__result_info *ri, int bpc)
5672 {
5673     int pixelCount;
5674     int channelCount, compression;
5675     int channel, i;
5676     int bitdepth;
5677     int w, h;
5678     stbi_uc *out;
5679     STBI_NOTUSED(ri);
5680
5681     // Check identifier
5682     if (stbi__get32be(s) != 0x38425053) // "8BPS"
5683         return stbi__errpuc("not PSD", "Corrupt PSD image");
5684
5685     // Check file type version.
5686     if (stbi__get16be(s) != 1)
5687         return stbi__errpuc("wrong version", "Unsupported version of PSD image");
5688
5689     // Skip 6 reserved bytes.
5690     stbi__skip(s, 6);
5691
5692     // Read the number of channels (R, G, B, A, etc).
5693     channelCount = stbi__get16be(s);
5694     if (channelCount < 0 || channelCount > 16)
5695         return stbi__errpuc("wrong channel count", "Unsupported number of channels in PSD image");
5696
5697     // Read the rows and columns of the image.
5698     h = stbi__get32be(s);
5699     w = stbi__get32be(s);
5700
5701     // Make sure the depth is 8 bits.
5702     bitdepth = stbi__get16be(s);
5703     if (bitdepth != 8 && bitdepth != 16)
5704         return stbi__errpuc("unsupported bit depth", "PSD bit depth is not 8 or 16 bit");
5705
5706     // Make sure the color mode is RGB.
5707     // Valid options are:
5708     // 0: Bitmap
5709     // 1: Grayscale
5710     // 2: Indexed color
5711     // 3: RGB color
5712     // 4: CMYK color
5713     // 7: Multichannel
5714     // 8: Duotone
5715     // 9: Lab color
5716     if (stbi__get16be(s) != 3)
5717         return stbi__errpuc("wrong color format", "PSD is not in RGB color format");
5718
5719     // Skip the Mode Data. (It's the palette for indexed color; other info for other modes.)
5720     stbi__skip(s, stbi__get32be(s));
5721
5722     // Skip the image resources. (resolution, pen tool paths, etc)
5723     stbi__skip(s, stbi__get32be(s));
5724
5725     // Skip the reserved data.
5726     stbi__skip(s, stbi__get32be(s));
5727
5728     // Find out if the data is compressed.
5729     // Known values:
5730     // 0: no compression
5731     // 1: RLE compressed
5732     compression = stbi__get16be(s);
5733     if (compression > 1)
5734         return stbi__errpuc("bad compression", "PSD has an unknown compression format");
5735
5736     // Check size

```

```

5737     if (!stbi__mad3sizes_valid(4, w, h, 0))
5738         return stbi__errpuc("too large", "Corrupt PSD");
5739
5740     // Create the destination image.
5741
5742     if (!compression && bitdepth == 16 && bpc == 16) {
5743         out = (stbi_uc *)stbi__malloc_mad3(8, w, h, 0);
5744         ri->bits_per_channel = 16;
5745     }
5746     else
5747         out = (stbi_uc *)stbi__malloc(4 * w*h);
5748
5749     if (!out) return stbi__errpuc("outofmem", "Out of memory");
5750     pixelCount = w*h;
5751
5752     // Initialize the data to zero.
5753     //memset( out, 0, pixelCount * 4 );
5754
5755     // Finally, the image data.
5756     if (compression) {
5757         // RLE as used by .PSD and .TIFF
5758         // Loop until you get the number of unpacked bytes you are expecting:
5759         //     Read the next source byte into n.
5760         //     If n is between 0 and 127 inclusive, copy the next n+1 bytes literally.
5761         //     Else if n is between -127 and -1 inclusive, copy the next byte -n+1 times.
5762         //     Else if n is 128, noop.
5763         // Endloop
5764
5765         // The RLE-compressed data is preceeded by a 2-byte data count for each row in the data,
5766         // which we're going to just skip.
5767         stbi__skip(s, h * channelCount * 2);
5768
5769         // Read the RLE data by channel.
5770         for (channel = 0; channel < 4; channel++) {
5771             stbi_uc *p;
5772
5773             p = out + channel;
5774             if (channel >= channelCount) {
5775                 // Fill this channel with default data.
5776                 for (i = 0; i < pixelCount; i++, p += 4)
5777                     *p = (channel == 3 ? 255 : 0);
5778             }
5779             else {
5780                 // Read the RLE data.
5781                 if (!stbi__psd_decode_rle(s, p, pixelCount)) {
5782                     STBI_FREE(out);
5783                     return stbi__errpuc("corrupt", "bad RLE data");
5784                 }
5785             }
5786         }
5787     }
5788     else {
5789         // We're at the raw image data. It's each channel in order (Red, Green, Blue, Alpha, ...)
5790         // where each channel consists of an 8-bit (or 16-bit) value for each pixel in the image.
5791
5792         // Read the data by channel.
5793         for (channel = 0; channel < 4; channel++) {
5794             if (channel >= channelCount) {
5795                 // Fill this channel with default data.
5796                 if (bitdepth == 16 && bpc == 16) {
5797                     stbi__uint16 *q = ((stbi__uint16 *)out) + channel;
5798                     stbi__uint16 val = channel == 3 ? 65535 : 0;
5799                     for (i = 0; i < pixelCount; i++, q += 4)
5800                         *q = val;
5801                 }
5802                 else {
5803                     stbi_uc *p = out + channel;
5804                     stbi_uc val = channel == 3 ? 255 : 0;
5805                     for (i = 0; i < pixelCount; i++, p += 4)
5806                         *p = val;
5807                 }
5808             }
5809             else {
5810                 if (ri->bits_per_channel == 16) { // output bpc
5811                     stbi__uint16 *q = ((stbi__uint16 *)out) + channel;
5812                     for (i = 0; i < pixelCount; i++, q += 4)
5813                         *q = (stbi__uint16)stbi__get16be(s);
5814                 }
5815                 else {
5816                     stbi_uc *p = out + channel;
5817                     if (bitdepth == 16) { // input bpc
5818                         for (i = 0; i < pixelCount; i++, p += 4)
5819                             *p = (stbi_uc)(stbi__get16be(s) >> 8);
5820                     }
5821                     else {
5822                         for (i = 0; i < pixelCount; i++, p += 4)

```



```

5824             *p = stbi__get8(s);
5825         }
5826     }
5827 }
5828 }
5829 }
5830
5831 // remove weird white matte from PSD
5832 if (channelCount >= 4) {
5833     if (ri->bits_per_channel == 16) {
5834         for (i = 0; i < w*h; ++i) {
5835             stbi_uint16 *pixel = (stbi_uint16 *)out + 4 * i;
5836             if (pixel[3] != 0 && pixel[3] != 65535) {
5837                 float a = pixel[3] / 65535.0f;
5838                 float ra = 1.0f / a;
5839                 float inv_a = 65535.0f * (1 - ra);
5840                 pixel[0] = (stbi_uint16)(pixel[0] * ra + inv_a);
5841                 pixel[1] = (stbi_uint16)(pixel[1] * ra + inv_a);
5842                 pixel[2] = (stbi_uint16)(pixel[2] * ra + inv_a);
5843             }
5844         }
5845     }
5846     else {
5847         for (i = 0; i < w*h; ++i) {
5848             unsigned char *pixel = out + 4 * i;
5849             if (pixel[3] != 0 && pixel[3] != 255) {
5850                 float a = pixel[3] / 255.0f;
5851                 float ra = 1.0f / a;
5852                 float inv_a = 255.0f * (1 - ra);
5853                 pixel[0] = (unsigned char)(pixel[0] * ra + inv_a);
5854                 pixel[1] = (unsigned char)(pixel[1] * ra + inv_a);
5855                 pixel[2] = (unsigned char)(pixel[2] * ra + inv_a);
5856             }
5857         }
5858     }
5859 }
5860
5861 // convert to desired output format
5862 if (req_comp && req_comp != 4) {
5863     if (ri->bits_per_channel == 16)
5864         out = (stbi_uc *)stbi__convert_format16((stbi_uint16 *)out, 4, req_comp, w, h);
5865     else
5866         out = stbi__convert_format(out, 4, req_comp, w, h);
5867     if (out == NULL) return out; // stbi__convert_format frees input on failure
5868 }
5869
5870 if (comp) *comp = 4;
5871 *y = h;
5872 *x = w;
5873
5874 return out;
5875 }
5876 #endif
5877
5878 // *****
5879 // Softimage PIC loader
5880 // by Tom Seddon
5881 //
5882 // See http://softimage.wiki.softimage.com/index.php/INFO:\_PIC\_file\_format
5883 // See http://ozviz.wasp.uwa.edu.au/~pbourke/dataformats/softimagepic/
5884
5885 #ifndef STBI_NO_PIC
5886 static int stbi__pic_is4(stbi__context *s, const char *str)
5887 {
5888     int i;
5889     for (i = 0; i < 4; ++i)
5890         if (stbi__get8(s) != (stbi_uc)str[i])
5891             return 0;
5892     return 1;
5893 }
5894
5895 static int stbi__pic_test_core(stbi__context *s)
5896 {
5897     int i;
5898     if (!stbi__pic_is4(s, "\x53\x80\xF6\x34"))
5899         return 0;
5900     for (i = 0; i < 84; ++i)
5901         stbi__get8(s);
5902     if (!stbi__pic_is4(s, "PICT"))
5903         return 0;
5904     return 1;
5905 }
5906 }

```

```

5911
5912 typedef struct
5913 {
5914     stbi_uc size, type, channel;
5915 } stbi__pic_packet;
5916
5917 static stbi_uc *stbi__readval(stbi__context *s, int channel, stbi_uc *dest)
5918 {
5919     int mask = 0x80, i;
5920
5921     for (i = 0; i < 4; ++i, mask >= 1) {
5922         if (channel & mask) {
5923             if (stbi__at_eof(s)) return stbi__errpuc("bad file", "PIC file too short");
5924             dest[i] = stbi__get8(s);
5925         }
5926     }
5927
5928     return dest;
5929 }
5930
5931 static void stbi__copyval(int channel, stbi_uc *dest, const stbi_uc *src)
5932 {
5933     int mask = 0x80, i;
5934
5935     for (i = 0; i < 4; ++i, mask >= 1)
5936         if (channel & mask)
5937             dest[i] = src[i];
5938 }
5939
5940 static stbi_uc *stbi__pic_load_core(stbi__context *s, int width, int height, int *comp, stbi_uc
    *result)
5941 {
5942     int act_comp = 0, num_packets = 0, y, chained;
5943     stbi__pic_packet packets[10];
5944
5945     // this will (should...) cater for even some bizarre stuff like having data
5946     // for the same channel in multiple packets.
5947     do {
5948         stbi__pic_packet *packet;
5949
5950         if (num_packets == sizeof(packets) / sizeof(packets[0]))
5951             return stbi__errpuc("bad format", "too many packets");
5952
5953         packet = &packets[num_packets++];
5954
5955         chained = stbi__get8(s);
5956         packet->size = stbi__get8(s);
5957         packet->type = stbi__get8(s);
5958         packet->channel = stbi__get8(s);
5959
5960         act_comp |= packet->channel;
5961
5962         if (stbi__at_eof(s)) return stbi__errpuc("bad file", "file too short (reading
            packets)");
5963         if (packet->size != 8) return stbi__errpuc("bad format", "packet isn't 8bpp");
5964     } while (chained);
5965
5966     *comp = (act_comp & 0x10 ? 4 : 3); // has alpha channel?
5967
5968     for (y = 0; y < height; ++y) {
5969         int packet_idx;
5970
5971         for (packet_idx = 0; packet_idx < num_packets; ++packet_idx) {
5972             stbi__pic_packet *packet = &packets[packet_idx];
5973             stbi_uc *dest = result + y*width * 4;
5974
5975             switch (packet->type) {
5976             default:
5977                 return stbi__errpuc("bad format", "packet has bad compression type");
5978
5979             case 0: //uncompressed
5980                 int x;
5981
5982                 for (x = 0; x < width; ++x, dest += 4)
5983                     if (!stbi__readval(s, packet->channel, dest))
5984                         return 0;
5985                 break;
5986             }
5987
5988             case 1: //Pure RLE
5989             {
5990                 int left = width, i;
5991
5992                 while (left > 0) {
5993                     stbi_uc count, value[4];
5994
5995                     count = stbi__get8(s);

```

```

5996         if (stbi__at_eof(s)) return stbi__errpuc("bad file", "file too short (pure read
count)");
5997
5998         if (count > left)
5999             count = (stbi_uc)left;
6000
6001         if (!stbi__readval(s, packet->channel, value)) return 0;
6002
6003         for (i = 0; i<count; ++i, dest += 4)
6004             stbi__copyval(packet->channel, dest, value);
6005         left -= count;
6006     }
6007 }
6008 break;
6009
6010 case 2: { //Mixed RLE
6011     int left = width;
6012     while (left>0) {
6013         int count = stbi__get8(s), i;
6014         if (stbi__at_eof(s)) return stbi__errpuc("bad file", "file too short (mixed read
count)");
6015
6016         if (count >= 128) { // Repeated
6017             stbi_uc value[4];
6018
6019             if (count == 128)
6020                 count = stbi__get16be(s);
6021             else
6022                 count -= 127;
6023             if (count > left)
6024                 return stbi__errpuc("bad file", "scanline overrun");
6025
6026             if (!stbi__readval(s, packet->channel, value))
6027                 return 0;
6028
6029             for (i = 0; i<count; ++i, dest += 4)
6030                 stbi__copyval(packet->channel, dest, value);
6031         }
6032         else { // Raw
6033             ++count;
6034             if (count>left) return stbi__errpuc("bad file", "scanline overrun");
6035
6036             for (i = 0; i<count; ++i, dest += 4)
6037                 if (!stbi__readval(s, packet->channel, dest))
6038                     return 0;
6039         }
6040         left -= count;
6041     }
6042     break;
6043 }
6044 }
6045 }
6046 }
6047
6048 return result;
6049 }
6050
6051 static void stbi__pic_load(stbi__context *s, int *px, int *py, int *comp, int req_comp,
stbi__result_info *ri)
6052 {
6053     stbi_uc *result;
6054     int i, x, y;
6055     STBI_NOTUSED(ri);
6056
6057     for (i = 0; i<92; ++i)
6058         stbi__get8(s);
6059
6060     x = stbi__get16be(s);
6061     y = stbi__get16be(s);
6062     if (stbi__at_eof(s)) return stbi__errpuc("bad file", "file too short (pic header)");
6063     if (!stbi__mad3sizes_valid(x, y, 4, 0)) return stbi__errpuc("too large", "PIC image too large to
decode");
6064
6065     stbi__get32be(s); //skip 'ratio'
6066     stbi__get16be(s); //skip 'fields'
6067     stbi__get16be(s); //skip 'pad'
6068
6069     // intermediate buffer is RGBA
6070     result = (stbi_uc *)stbi__malloc_mad3(x, y, 4, 0);
6071     memset(result, 0xff, x*y * 4);
6072
6073     if (!stbi__pic_load_core(s, x, y, comp, result)) {
6074         STBI_FREE(result);
6075         result = 0;
6076     }
6077     *px = x;
6078     *py = y;

```

```

6079     if (req_comp == 0) req_comp = *comp;
6080     result = stbi__convert_format(result, 4, req_comp, x, y);
6081
6082     return result;
6083 }
6084
6085 static int stbi__pic_test(stbi__context *s)
6086 {
6087     int r = stbi__pic_test_core(s);
6088     stbi__rewind(s);
6089     return r;
6090 }
6091 #endif
6092
6093 // *****
6094 // GIF loader -- public domain by Jean-Marc Lienher -- simplified/shrunk by stb
6095
6096 #ifndef STBI_NO_GIF
6097 typedef struct
6098 {
6099     stbi__int16 prefix;
6100     stbi_uc first;
6101     stbi_uc suffix;
6102 } stbi__gif_lzw;
6103
6104 typedef struct
6105 {
6106     int w, h;
6107     stbi_uc *out, *old_out;           // output buffer (always 4 components)
6108     int flags, bgindex, ratio, transparent, eflags, delay;
6109     stbi_uc pal[256][4];
6110     stbi_uc lpal[256][4];
6111     stbi__gif_lzw codes[4096];
6112     stbi_uc *color_table;
6113     int parse, step;
6114     int lflags;
6115     int start_x, start_y;
6116     int max_x, max_y;
6117     int cur_x, cur_y;
6118     int line_size;
6119 } stbi__gif;
6120
6121 static int stbi__gif_test_raw(stbi__context *s)
6122 {
6123     int sz;
6124     if (stbi__get8(s) != 'G' || stbi__get8(s) != 'I' || stbi__get8(s) != 'F' || stbi__get8(s) != '8')
6125         return 0;
6126     sz = stbi__get8(s);
6127     if (sz != '9' && sz != '7') return 0;
6128     if (stbi__get8(s) != 'a') return 0;
6129     return 1;
6130 }
6131
6132 static int stbi__gif_test(stbi__context *s)
6133 {
6134     int r = stbi__gif_test_raw(s);
6135     stbi__rewind(s);
6136     return r;
6137 }
6138
6139 static void stbi__gif_parse_colortable(stbi__context *s, stbi_uc pal[256][4], int num_entries, int transp)
6140 {
6141     int i;
6142     for (i = 0; i < num_entries; ++i) {
6143         pal[i][2] = stbi__get8(s);
6144         pal[i][1] = stbi__get8(s);
6145         pal[i][0] = stbi__get8(s);
6146         pal[i][3] = transp == i ? 0 : 255;
6147     }
6148 }
6149
6150 static int stbi__gif_header(stbi__context *s, stbi__gif *g, int *comp, int is_info)
6151 {
6152     stbi_uc version;
6153     if (stbi__get8(s) != 'G' || stbi__get8(s) != 'I' || stbi__get8(s) != 'F' || stbi__get8(s) != '8')
6154         return stbi__err("not GIF", "Corrupt GIF");
6155
6156     version = stbi__get8(s);
6157     if (version != '7' && version != '9') return stbi__err("not GIF", "Corrupt GIF");
6158     if (stbi__get8(s) != 'a') return stbi__err("not GIF", "Corrupt GIF");
6159
6160     stbi__g_failure_reason = "";
6161     g->w = stbi__get16le(s);
6162     g->h = stbi__get16le(s);
6163     g->flags = stbi__get8(s);
6164     g->bgindex = stbi__get8(s);

```

```

6164     g->ratio = stbi__get8(s);
6165     g->transparent = -1;
6166
6167     if (comp != 0) *comp = 4; // can't actually tell whether it's 3 or 4 until we parse the comments
6168
6169     if (is_info) return 1;
6170
6171     if (g->flags & 0x80)
6172         stbi__gif_parse_colortable(s, g->pal, 2 « (g->flags & 7), -1);
6173
6174     return 1;
6175 }
6176
6177 static int stbi__gif_info_raw(stbi__context *s, int *x, int *y, int *comp)
6178 {
6179     stbi__gif* g = (stbi__gif*)stbi__malloc(sizeof(stbi__gif));
6180     if (!stbi__gif_header(s, g, comp, 1)) {
6181         STBI_FREE(g);
6182         stbi__rewind(s);
6183         return 0;
6184     }
6185     if (x) *x = g->w;
6186     if (y) *y = g->h;
6187     STBI_FREE(g);
6188     return 1;
6189 }
6190
6191 static void stbi__out_gif_code(stbi__gif *g, stbi__uint16 code)
6192 {
6193     stbi_uc *p, *c;
6194
6195     // recurse to decode the prefixes, since the linked-list is backwards,
6196     // and working backwards through an interleaved image would be nasty
6197     if (g->codes[code].prefix >= 0)
6198         stbi__out_gif_code(g, g->codes[code].prefix);
6199
6200     if (g->cur_y >= g->max_y) return;
6201
6202     p = &g->out[g->cur_x + g->cur_y];
6203     c = &g->color_table[g->codes[code].suffix * 4];
6204
6205     if (c[3] >= 128) {
6206         p[0] = c[2];
6207         p[1] = c[1];
6208         p[2] = c[0];
6209         p[3] = c[3];
6210     }
6211     g->cur_x += 4;
6212
6213     if (g->cur_x >= g->max_x) {
6214         g->cur_x = g->start_x;
6215         g->cur_y += g->step;
6216
6217         while (g->cur_y >= g->max_y && g->parse > 0) {
6218             g->step = (1 « g->parse) * g->line_size;
6219             g->cur_y = g->start_y + (g->step » 1);
6220             --g->parse;
6221         }
6222     }
6223 }
6224
6225 static stbi_uc *stbi__process_gif_raster(stbi__context *s, stbi__gif *g)
6226 {
6227     stbi_uc lzw_cs;
6228     stbi__int32 len, init_code;
6229     stbi__uint32 first;
6230     stbi__int32 codesize, codemask, avail, oldcode, bits, valid_bits, clear;
6231     stbi__gif_lzw *p;
6232
6233     lzw_cs = stbi__get8(s);
6234     if (lzw_cs > 12) return NULL;
6235     clear = 1 « lzw_cs;
6236     first = 1;
6237     codesize = lzw_cs + 1;
6238     codemask = (1 « codesize) - 1;
6239     bits = 0;
6240     valid_bits = 0;
6241     for (init_code = 0; init_code < clear; init_code++) {
6242         g->codes[init_code].prefix = -1;
6243         g->codes[init_code].first = (stbi_uc)init_code;
6244         g->codes[init_code].suffix = (stbi_uc)init_code;
6245     }
6246
6247     // support no starting clear code
6248     avail = clear + 2;
6249     oldcode = -1;
6250

```

```

6251     len = 0;
6252     for (;;) {
6253         if (valid_bits < codesize) {
6254             if (len == 0) {
6255                 len = stbi__get8(s); // start new block
6256                 if (len == 0)
6257                     return g->out;
6258             }
6259             --len;
6260             bits |= (stbi__int32)stbi__get8(s) << valid_bits;
6261             valid_bits += 8;
6262         }
6263         else {
6264             stbi__int32 code = bits & codemask;
6265             bits >>= codesize;
6266             valid_bits -= codesize;
6267             // @OPTIMIZE: is there some way we can accelerate the non-clear path?
6268             if (code == clear) { // clear code
6269                 codesize = lzw_cs + 1;
6270                 codemask = (1 << codesize) - 1;
6271                 avail = clear + 2;
6272                 oldcode = -1;
6273                 first = 0;
6274             }
6275             else if (code == clear + 1) { // end of stream code
6276                 stbi__skip(s, len);
6277                 while ((len = stbi__get8(s)) > 0)
6278                     stbi__skip(s, len);
6279                 return g->out;
6280             }
6281             else if (code <= avail) {
6282                 if (first) return stbi__errpuc("no clear code", "Corrupt GIF");
6283
6284                 if (oldcode >= 0) {
6285                     p = &g->codes[avail++];
6286                     if (avail > 4096) return stbi__errpuc("too many codes", "Corrupt GIF");
6287                     p->prefix = (stbi__int16)oldcode;
6288                     p->first = g->codes[oldcode].first;
6289                     p->suffix = (code == avail) ? p->first : g->codes[code].first;
6290                 }
6291                 else if (code == avail)
6292                     return stbi__errpuc("illegal code in raster", "Corrupt GIF");
6293
6294                 stbi__out_gif_code(g, (stbi__uint16)code);
6295
6296                 if ((avail & codemask) == 0 && avail <= 0x0FFF) {
6297                     codesize++;
6298                     codemask = (1 << codesize) - 1;
6299                 }
6300
6301                 oldcode = code;
6302             }
6303             else {
6304                 return stbi__errpuc("illegal code in raster", "Corrupt GIF");
6305             }
6306         }
6307     }
6308 }
6309
6310 static void stbi__fill_gif_background(stbi__gif *g, int x0, int y0, int x1, int y1)
6311 {
6312     int x, y;
6313     stbi_uc *c = g->pal[g->bgindex];
6314     for (y = y0; y < y1; y += 4 * g->w) {
6315         for (x = x0; x < x1; x += 4) {
6316             stbi_uc *p = &g->out[y + x];
6317             p[0] = c[2];
6318             p[1] = c[1];
6319             p[2] = c[0];
6320             p[3] = 0;
6321         }
6322     }
6323 }
6324
6325 // this function is designed to support animated gifs, although stb_image doesn't support it
6326 static stbi_uc *stbi__gif_load_next(stbi__context *s, stbi__gif *g, int *comp, int req_comp)
6327 {
6328     int i;
6329     stbi_uc *prev_out = 0;
6330
6331     if (g->out == 0 && !stbi__gif_header(s, g, comp, 0))
6332         return 0; // stbi__g_failure_reason set by stbi__gif_header
6333
6334     if (!stbi__mad3sizes_valid(g->w, g->h, 4, 0))
6335         return stbi__errpuc("too large", "GIF too large");
6336
6337     prev_out = g->out;

```

```

6338     g->out = (stbi_uc *)stbi__malloc_mad3(4, g->w, g->h, 0);
6339     if (g->out == 0) return stbi__errpuc("outofmem", "Out of memory");
6340
6341     switch ((g->eflags & 0x1C) >> 2) {
6342     case 0: // unspecified (also always used on 1st frame)
6343         stbi__fill_gif_background(g, 0, 0, 4 * g->w, 4 * g->w * g->h);
6344         break;
6345     case 1: // do not dispose
6346         if (prev_out) memcpy(g->out, prev_out, 4 * g->w * g->h);
6347         g->old_out = prev_out;
6348         break;
6349     case 2: // dispose to background
6350         if (prev_out) memcpy(g->out, prev_out, 4 * g->w * g->h);
6351         stbi__fill_gif_background(g, g->start_x, g->start_y, g->max_x, g->max_y);
6352         break;
6353     case 3: // dispose to previous
6354         if (g->old_out) {
6355             for (i = g->start_y; i < g->max_y; i += 4 * g->w)
6356                 memcpy(&g->out[i + g->start_x], &g->old_out[i + g->start_x], g->max_x - g->start_x);
6357         }
6358         break;
6359     }
6360
6361     for (;;) {
6362         switch (stbi__get8(s)) {
6363         case 0x2C: /* Image Descriptor */
6364             {
6365                 int prev_trans = -1;
6366                 stbi__int32 x, y, w, h;
6367                 stbi_uc *o;
6368
6369                 x = stbi__get16le(s);
6370                 y = stbi__get16le(s);
6371                 w = stbi__get16le(s);
6372                 h = stbi__get16le(s);
6373                 if (((x + w) > (g->w)) || ((y + h) > (g->h)))
6374                     return stbi__errpuc("bad Image Descriptor", "Corrupt GIF");
6375
6376                 g->line_size = g->w * 4;
6377                 g->start_x = x * 4;
6378                 g->start_y = y * g->line_size;
6379                 g->max_x = g->start_x + w * 4;
6380                 g->max_y = g->start_y + h * g->line_size;
6381                 g->cur_x = g->start_x;
6382                 g->cur_y = g->start_y;
6383
6384                 g->lflags = stbi__get8(s);
6385
6386                 if (g->lflags & 0x40) {
6387                     g->step = 8 * g->line_size; // first interlaced spacing
6388                     g->parse = 3;
6389                 }
6390                 else {
6391                     g->step = g->line_size;
6392                     g->parse = 0;
6393                 }
6394
6395                 if (g->lflags & 0x80) {
6396                     stbi__gif_parse_colortable(s, g->lpal, 2 << (g->lflags & 7), g->eflags & 0x01 ?
g->transparent : -1);
6397                     g->color_table = (stbi_uc *)g->lpal;
6398                 }
6399                 else if (g->flags & 0x80) {
6400                     if (g->transparent >= 0 && (g->eflags & 0x01)) {
6401                         prev_trans = g->pal[g->transparent][3];
6402                         g->pal[g->transparent][3] = 0;
6403                     }
6404                     g->color_table = (stbi_uc *)g->pal;
6405                 }
6406                 else
6407                     return stbi__errpuc("missing color table", "Corrupt GIF");
6408
6409                 o = stbi__process_gif_raster(s, g);
6410                 if (o == NULL) return NULL;
6411
6412                 if (prev_trans != -1)
6413                     g->pal[g->transparent][3] = (stbi_uc)prev_trans;
6414
6415                 return o;
6416             }
6417
6418         case 0x21: /* Comment Extension.
6419             {
6420                 int len;
6421                 if (stbi__get8(s) == 0xF9) { // Graphic Control Extension.
6422                     len = stbi__get8(s);
6423                     if (len == 4) {

```

```

6424         g->eflags = stbi__get8(s);
6425         g->delay = stbi__get16le(s);
6426         g->transparent = stbi__get8(s);
6427     }
6428     else {
6429         stbi__skip(s, len);
6430         break;
6431     }
6432 }
6433 while ((len = stbi__get8(s)) != 0)
6434     stbi__skip(s, len);
6435 break;
6436 }
6437
6438 case 0x3B: // gif stream termination code
6439     return (stbi_uc *)s; // using 'l' causes warning on some compilers
6440
6441 default:
6442     return stbi__errpuc("unknown code", "Corrupt GIF");
6443 }
6444 }
6445
6446 STBI_NOTUSED(req_comp);
6447 }
6448
6449 static void *stbi__gif_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
        stbi__result_info *ri)
6450 {
6451     stbi_uc *u = 0;
6452     stbi__gif* g = (stbi__gif*)stbi__malloc(sizeof(stbi__gif));
6453     memset(g, 0, sizeof(*g));
6454     STBI_NOTUSED(ri);
6455
6456     u = stbi__gif_load_next(s, g, comp, req_comp);
6457     if (u == (stbi_uc *)s) u = 0; // end of animated gif marker
6458     if (u) {
6459         *x = g->w;
6460         *y = g->h;
6461         if (req_comp && req_comp != 4)
6462             u = stbi__convert_format(u, 4, req_comp, g->w, g->h);
6463     }
6464     else if (g->out)
6465         STBI_FREE(g->out);
6466     STBI_FREE(g);
6467     return u;
6468 }
6469
6470 static int stbi__gif_info(stbi__context *s, int *x, int *y, int *comp)
6471 {
6472     return stbi__gif_info_raw(s, x, y, comp);
6473 }
6474 #endif
6475
6476 // *****
6477 // Radiance RGBE HDR loader
6478 // originally by Nicolas Schulz
6479 #ifndef STBI_NO_HDR
6480 static int stbi__hdr_test_core(stbi__context *s, const char *signature)
6481 {
6482     int i;
6483     for (i = 0; signature[i]; ++i)
6484         if (stbi__get8(s) != signature[i])
6485             return 0;
6486     stbi__rewind(s);
6487     return 1;
6488 }
6489
6490 static int stbi__hdr_test(stbi__context* s)
6491 {
6492     int r = stbi__hdr_test_core(s, "##?RADIANCE\n");
6493     stbi__rewind(s);
6494     if (!r) {
6495         r = stbi__hdr_test_core(s, "##?RGBE\n");
6496         stbi__rewind(s);
6497     }
6498     return r;
6499 }
6500
6501 #define STBI__HDR_BUFLEN 1024
6502 static char *stbi__hdr_gettoken(stbi__context *z, char *buffer)
6503 {
6504     int len = 0;
6505     char c = '\0';
6506
6507     c = (char)stbi__get8(z);
6508
6509     while (!stbi__at_eof(z) && c != '\n') {

```



```

6510         buffer[len++] = c;
6511         if (len == STBI__HDR_BUFLEN - 1) {
6512             // flush to end of line
6513             while (!stbi__at_eof(z) && stbi__get8(z) != '\n')
6514                 ;
6515             break;
6516         }
6517         c = (char)stbi__get8(z);
6518     }
6519     buffer[len] = 0;
6520     return buffer;
6521 }
6522
6523
6524 static void stbi__hdr_convert(float *output, stbi_uc *input, int req_comp)
6525 {
6526     if (input[3] != 0) {
6527         float f1;
6528         // Exponent
6529         f1 = (float)ldexp(1.0f, input[3] - (int)(128 + 8));
6530         if (req_comp <= 2)
6531             output[0] = (input[0] + input[1] + input[2]) * f1 / 3;
6532         else {
6533             output[0] = input[0] * f1;
6534             output[1] = input[1] * f1;
6535             output[2] = input[2] * f1;
6536         }
6537         if (req_comp == 2) output[1] = 1;
6538         if (req_comp == 4) output[3] = 1;
6539     }
6540     else {
6541         switch (req_comp) {
6542             case 4: output[3] = 1; /* fallthrough */
6543             case 3: output[0] = output[1] = output[2] = 0;
6544                 break;
6545             case 2: output[1] = 1; /* fallthrough */
6546             case 1: output[0] = 0;
6547                 break;
6548         }
6549     }
6550 }
6551
6552 static float *stbi__hdr_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
6553     stbi__result_info *ri)
6554 {
6555     char buffer[STBI__HDR_BUFLEN];
6556     char *token;
6557     int valid = 0;
6558     int width, height;
6559     stbi_uc *scanline;
6560     float *hdr_data;
6561     int len;
6562     unsigned char count, value;
6563     int i, j, k, c1, c2, z;
6564     const char *headerToken;
6565     STBI_NOTUSED(ri);
6566
6567     // Check identifier
6568     headerToken = stbi__hdr_gettoken(s, buffer);
6569     if (strcmp(headerToken, "#?RADIANCE") != 0 && strcmp(headerToken, "#?RGBE") != 0)
6570         return stbi__errpf("not HDR", "Corrupt HDR image");
6571
6572     // Parse header
6573     for (;;) {
6574         token = stbi__hdr_gettoken(s, buffer);
6575         if (token[0] == 0) break;
6576         if (strcmp(token, "FORMAT=32-bit_rle_rgbe") == 0) valid = 1;
6577     }
6578
6579     if (!valid) return stbi__errpf("unsupported format", "Unsupported HDR format");
6580
6581     // Parse width and height
6582     // can't use sscanf() if we're not using stdio!
6583     token = stbi__hdr_gettoken(s, buffer);
6584     if (strncmp(token, "-Y ", 3)) return stbi__errpf("unsupported data layout", "Unsupported HDR
6585     format");
6586     token += 3;
6587     height = (int)strtol(token, &token, 10);
6588     while (*token == ' ') ++token;
6589     token = stbi__hdr_gettoken(s, buffer);
6590     if (strncmp(token, "+X ", 3)) return stbi__errpf("unsupported data layout", "Unsupported HDR
6591     format");
6592     token += 3;
6593     width = (int)strtol(token, NULL, 10);
6594
6595     *x = width;
6596     *y = height;

```

```

6594     if (comp) *comp = 3;
6595     if (req_comp == 0) req_comp = 3;
6596
6597     if (!stbi__mad4sizes_valid(width, height, req_comp, sizeof(float), 0))
6598         return stbi__errpf("too large", "HDR image is too large");
6599
6600     // Read data
6601     hdr_data = (float *)stbi__malloc_mad4(width, height, req_comp, sizeof(float), 0);
6602     if (!hdr_data)
6603         return stbi__errpf("outofmem", "Out of memory");
6604
6605     // Load image data
6606     // image data is stored as some number of sca
6607     if (width < 8 || width >= 32768) {
6608         // Read flat data
6609         for (j = 0; j < height; ++j) {
6610             for (i = 0; i < width; ++i) {
6611                 stbi_uc rgb[4];
6612                 main_decode_loop:
6613                     stbi__getn(s, rgb, 4);
6614                     stbi__hdr_convert(hdr_data + j * width * req_comp + i * req_comp, rgb, req_comp);
6615             }
6616         }
6617     }
6618     else {
6619         // Read RLE-encoded data
6620         scanline = NULL;
6621
6622         for (j = 0; j < height; ++j) {
6623             c1 = stbi__get8(s);
6624             c2 = stbi__get8(s);
6625             len = stbi__get8(s);
6626             if (c1 != 2 || c2 != 2 || (len & 0x80)) {
6627                 // not run-length encoded, so we have to actually use THIS data as a decoded
6628                 // pixel (note this can't be a valid pixel--one of RGB must be >= 128)
6629                 stbi_uc rgb[4];
6630                 rgb[0] = (stbi_uc)c1;
6631                 rgb[1] = (stbi_uc)c2;
6632                 rgb[2] = (stbi_uc)len;
6633                 rgb[3] = (stbi_uc)stbi__get8(s);
6634                 stbi__hdr_convert(hdr_data, rgb, req_comp);
6635                 i = 1;
6636                 j = 0;
6637                 STBI_FREE(scanline);
6638                 goto main_decode_loop; // yes, this makes no sense
6639             }
6640             len <<= 8;
6641             len |= stbi__get8(s);
6642             if (len != width) { STBI_FREE(hdr_data); STBI_FREE(scanline); return stbi__errpf("invalid
6643 decoded scanline length", "corrupt HDR"); }
6644             if (scanline == NULL) {
6645                 scanline = (stbi_uc *)stbi__malloc_mad2(width, 4, 0);
6646                 if (!scanline) {
6647                     STBI_FREE(hdr_data);
6648                     return stbi__errpf("outofmem", "Out of memory");
6649                 }
6650             }
6651             for (k = 0; k < 4; ++k) {
6652                 int nleft;
6653                 i = 0;
6654                 while ((nleft = width - i) > 0) {
6655                     count = stbi__get8(s);
6656                     if (count > 128) {
6657                         // Run
6658                         value = stbi__get8(s);
6659                         count -= 128;
6660                         if (count > nleft) { STBI_FREE(hdr_data); STBI_FREE(scanline); return
6661 stbi__errpf("corrupt", "bad RLE data in HDR"); }
6662                         for (z = 0; z < count; ++z)
6663                             scanline[i++ * 4 + k] = value;
6664                     }
6665                     else {
6666                         // Dump
6667                         if (count > nleft) { STBI_FREE(hdr_data); STBI_FREE(scanline); return
6668 stbi__errpf("corrupt", "bad RLE data in HDR"); }
6669                         for (z = 0; z < count; ++z)
6670                             scanline[i++ * 4 + k] = stbi__get8(s);
6671                     }
6672                 }
6673                 for (i = 0; i < width; ++i)
6674                     stbi__hdr_convert(hdr_data + (j*width + i)*req_comp, scanline + i * 4, req_comp);
6675             }
6676             if (scanline)
6677                 STBI_FREE(scanline);
6678         }

```

```

6678
6679     return hdr_data;
6680 }
6681
6682 static int stbi__hdr_info(stbi__context *s, int *x, int *y, int *comp)
6683 {
6684     char buffer[STBI__HDR_BUFLEN];
6685     char *token;
6686     int valid = 0;
6687
6688     if (stbi__hdr_test(s) == 0) {
6689         stbi__rewind(s);
6690         return 0;
6691     }
6692
6693     for (;;) {
6694         token = stbi__hdr_gettoken(s, buffer);
6695         if (token[0] == 0) break;
6696         if (strcmp(token, "FORMAT=32-bit_rle_rgbe") == 0) valid = 1;
6697     }
6698
6699     if (!valid) {
6700         stbi__rewind(s);
6701         return 0;
6702     }
6703     token = stbi__hdr_gettoken(s, buffer);
6704     if (strncmp(token, "-Y ", 3)) {
6705         stbi__rewind(s);
6706         return 0;
6707     }
6708     token += 3;
6709     *y = (int)strtol(token, &token, 10);
6710     while (*token == ' ') ++token;
6711     if (strncmp(token, "+X ", 3)) {
6712         stbi__rewind(s);
6713         return 0;
6714     }
6715     token += 3;
6716     *x = (int)strtol(token, NULL, 10);
6717     *comp = 3;
6718     return 1;
6719 }
6720 #endif // STBI_NO_HDR
6721
6722 #ifndef STBI_NO_BMP
6723 static int stbi__bmp_info(stbi__context *s, int *x, int *y, int *comp)
6724 {
6725     void *p;
6726     stbi__bmp_data info;
6727
6728     info.all_a = 255;
6729     p = stbi__bmp_parse_header(s, &info);
6730     stbi__rewind(s);
6731     if (p == NULL)
6732         return 0;
6733     *x = s->img_x;
6734     *y = s->img_y;
6735     *comp = info.ma ? 4 : 3;
6736     return 1;
6737 }
6738 #endif
6739
6740 #ifndef STBI_NO_PSD
6741 static int stbi__psd_info(stbi__context *s, int *x, int *y, int *comp)
6742 {
6743     int channelCount;
6744     if (stbi__get32be(s) != 0x38425053) {
6745         stbi__rewind(s);
6746         return 0;
6747     }
6748     if (stbi__get16be(s) != 1) {
6749         stbi__rewind(s);
6750         return 0;
6751     }
6752     stbi__skip(s, 6);
6753     channelCount = stbi__get16be(s);
6754     if (channelCount < 0 || channelCount > 16) {
6755         stbi__rewind(s);
6756         return 0;
6757     }
6758     *y = stbi__get32be(s);
6759     *x = stbi__get32be(s);
6760     if (stbi__get16be(s) != 8) {
6761         stbi__rewind(s);
6762         return 0;
6763     }
6764     if (stbi__get16be(s) != 3) {

```

```

6765         stbi__rewind(s);
6766         return 0;
6767     }
6768     *comp = 4;
6769     return 1;
6770 }
6771 #endif
6772
6773 #ifndef STBI_NO_PIC
6774 static int stbi__pic_info(stbi__context *s, int *x, int *y, int *comp)
6775 {
6776     int act_comp = 0, num_packets = 0, chained;
6777     stbi__pic_packet packets[10];
6778
6779     if (!stbi__pic_is4(s, "\x53\x80\xF6\x34")) {
6780         stbi__rewind(s);
6781         return 0;
6782     }
6783
6784     stbi__skip(s, 88);
6785
6786     *x = stbi__get16be(s);
6787     *y = stbi__get16be(s);
6788     if (stbi__at_eof(s)) {
6789         stbi__rewind(s);
6790         return 0;
6791     }
6792     if ((*x) != 0 && (1 << 28) / (*x) < (*y)) {
6793         stbi__rewind(s);
6794         return 0;
6795     }
6796
6797     stbi__skip(s, 8);
6798
6799     do {
6800         stbi__pic_packet *packet;
6801
6802         if (num_packets == sizeof(packets) / sizeof(packets[0]))
6803             return 0;
6804
6805         packet = &packets[num_packets++];
6806         chained = stbi__get8(s);
6807         packet->size = stbi__get8(s);
6808         packet->type = stbi__get8(s);
6809         packet->channel = stbi__get8(s);
6810         act_comp |= packet->channel;
6811
6812         if (stbi__at_eof(s)) {
6813             stbi__rewind(s);
6814             return 0;
6815         }
6816         if (packet->size != 8) {
6817             stbi__rewind(s);
6818             return 0;
6819         }
6820     } while (chained);
6821
6822     *comp = (act_comp & 0x10 ? 4 : 3);
6823
6824     return 1;
6825 }
6826 #endif
6827
6828 // *****
6829 // Portable Gray Map and Portable Pixel Map loader
6830 // by Ken Miller
6831 //
6832 // PGM: http://netpbm.sourceforge.net/doc/pgm.html
6833 // PPM: http://netpbm.sourceforge.net/doc/ppm.html
6834 //
6835 // Known limitations:
6836 //   Does not support comments in the header section
6837 //   Does not support ASCII image data (formats P2 and P3)
6838 //   Does not support 16-bit-per-channel
6839
6840 #ifndef STBI_NO_PNM
6841
6842 static int stbi__pnm_test(stbi__context *s)
6843 {
6844     char p, t;
6845     p = (char)stbi__get8(s);
6846     t = (char)stbi__get8(s);
6847     if (p != 'P' || (t != '5' && t != '6')) {
6848         stbi__rewind(s);
6849         return 0;
6850     }
6851     return 1;

```

```

6852 }
6853
6854 static void stbi_pnm_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
    stbi__result_info *ri)
6855 {
6856     stbi_uc *out;
6857     STBI_NOTUSED(ri);
6858
6859     if (!stbi_pnm_info(s, (int *)&s->img_x, (int *)&s->img_y, (int *)&s->img_n))
6860         return 0;
6861
6862     *x = s->img_x;
6863     *y = s->img_y;
6864     *comp = s->img_n;
6865
6866     if (!stbi__mad3sizes_valid(s->img_n, s->img_x, s->img_y, 0))
6867         return stbi__errpuc("too large", "PNM too large");
6868
6869     out = (stbi_uc *)stbi__malloc_mad3(s->img_n, s->img_x, s->img_y, 0);
6870     if (!out) return stbi__errpuc("outofmem", "Out of memory");
6871     stbi__getn(s, out, s->img_n * s->img_x * s->img_y);
6872
6873     if (req_comp && req_comp != s->img_n) {
6874         out = stbi__convert_format(out, s->img_n, req_comp, s->img_x, s->img_y);
6875         if (out == NULL) return out; // stbi__convert_format frees input on failure
6876     }
6877     return out;
6878 }
6879
6880 static int stbi_pnm_isspace(char c)
6881 {
6882     return c == ' ' || c == '\t' || c == '\n' || c == '\v' || c == '\f' || c == '\r';
6883 }
6884
6885 static void stbi_pnm_skip_whitespace(stbi__context *s, char *c)
6886 {
6887     for (;;) {
6888         while (!stbi__at_eof(s) && stbi_pnm_isspace(*c))
6889             *c = (char)stbi__get8(s);
6890
6891         if (stbi__at_eof(s) || *c != '#')
6892             break;
6893
6894         while (!stbi__at_eof(s) && *c != '\n' && *c != '\r')
6895             *c = (char)stbi__get8(s);
6896     }
6897 }
6898
6899 static int stbi_pnm_isdigit(char c)
6900 {
6901     return c >= '0' && c <= '9';
6902 }
6903
6904 static int stbi_pnm_getinteger(stbi__context *s, char *c)
6905 {
6906     int value = 0;
6907
6908     while (!stbi__at_eof(s) && stbi_pnm_isdigit(*c)) {
6909         value = value * 10 + (*c - '0');
6910         *c = (char)stbi__get8(s);
6911     }
6912
6913     return value;
6914 }
6915
6916 static int stbi_pnm_info(stbi__context *s, int *x, int *y, int *comp)
6917 {
6918     int maxv;
6919     char c, p, t;
6920
6921     stbi__rewind(s);
6922
6923     // Get identifier
6924     p = (char)stbi__get8(s);
6925     t = (char)stbi__get8(s);
6926     if (p != 'P' || (t != '5' && t != '6')) {
6927         stbi__rewind(s);
6928         return 0;
6929     }
6930
6931     *comp = (t == '6') ? 3 : 1; // '5' is 1-component .pgm; '6' is 3-component .ppm
6932
6933     c = (char)stbi__get8(s);
6934     stbi_pnm_skip_whitespace(s, &c);
6935
6936     *x = stbi_pnm_getinteger(s, &c); // read width
6937     stbi_pnm_skip_whitespace(s, &c);

```

```

6938
6939     *y = stbi__pnm_getinteger(s, &c); // read height
6940     stbi__pnm_skip_whitespace(s, &c);
6941
6942     maxv = stbi__pnm_getinteger(s, &c); // read max value
6943
6944     if (maxv > 255)
6945         return stbi__err("max value > 255", "PPM image not 8-bit");
6946     else
6947         return 1;
6948 }
6949 #endif
6950
6951 static int stbi__info_main(stbi__context *s, int *x, int *y, int *comp)
6952 {
6953     #ifndef STBI_NO_JPEG
6954         if (stbi__jpeg_info(s, x, y, comp)) return 1;
6955     #endif
6956
6957     #ifndef STBI_NO_PNG
6958         if (stbi__png_info(s, x, y, comp)) return 1;
6959     #endif
6960
6961     #ifndef STBI_NO_GIF
6962         if (stbi__gif_info(s, x, y, comp)) return 1;
6963     #endif
6964
6965     #ifndef STBI_NO_BMP
6966         if (stbi__bmp_info(s, x, y, comp)) return 1;
6967     #endif
6968
6969     #ifndef STBI_NO_PSD
6970         if (stbi__psd_info(s, x, y, comp)) return 1;
6971     #endif
6972
6973     #ifndef STBI_NO_PIC
6974         if (stbi__pic_info(s, x, y, comp)) return 1;
6975     #endif
6976
6977     #ifndef STBI_NO_PNM
6978         if (stbi__pnm_info(s, x, y, comp)) return 1;
6979     #endif
6980
6981     #ifndef STBI_NO_HDR
6982         if (stbi__hdr_info(s, x, y, comp)) return 1;
6983     #endif
6984
6985     // test tga last because it's a crappy test!
6986     #ifndef STBI_NO_TGA
6987         if (stbi__tga_info(s, x, y, comp))
6988             return 1;
6989     #endif
6990     return stbi__err("unknown image type", "Image not of any known type, or corrupt");
6991 }
6992
6993 #ifndef STBI_NO_STDIO
6994 STBIDEF int stbi_info(char const *filename, int *x, int *y, int *comp)
6995 {
6996     FILE *f = stbi__fopen(filename, "rb");
6997     int result;
6998     if (!f) return stbi__err("can't fopen", "Unable to open file");
6999     result = stbi_info_from_file(f, x, y, comp);
7000     fclose(f);
7001     return result;
7002 }
7003
7004 STBIDEF int stbi_info_from_file(FILE *f, int *x, int *y, int *comp)
7005 {
7006     int r;
7007     stbi__context s;
7008     long pos = ftell(f);
7009     stbi__start_file(&s, f);
7010     r = stbi__info_main(&s, x, y, comp);
7011     fseek(f, pos, SEEK_SET);
7012     return r;
7013 }
7014 #endif // !STBI_NO_STDIO
7015
7016 STBIDEF int stbi_info_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp)
7017 {
7018     stbi__context s;
7019     stbi__start_mem(&s, buffer, len);
7020     return stbi__info_main(&s, x, y, comp);
7021 }
7022
7023 STBIDEF int stbi_info_from_callbacks(stbi_io_callbacks const *c, void *user, int *x, int *y, int *comp)
7024 {

```

```

7025     stbi__context s;
7026     stbi__start_callbacks(&s, (stbi_io_callbacks *)c, user);
7027     return stbi__info_main(&s, x, y, comp);
7028 }
7029
7030 #endif // STB_IMAGE_IMPLEMENTATION
7031
7032 /*
7033 revision history:
7034 2.13 (2016-11-29) add 16-bit API, only supported for PNG right now
7035 2.12 (2016-04-02) fix typo in 2.11 PSD fix that caused crashes
7036 2.11 (2016-04-02) allocate large structures on the stack
7037 remove white matting for transparent PSD
7038 fix reported channel count for PNG & BMP
7039 re-enable SSE2 in non-gcc 64-bit
7040 support RGB-formatted JPEG
7041 read 16-bit PNGs (only as 8-bit)
7042 2.10 (2016-01-22) avoid warning introduced in 2.09 by STBI_REALLOC_SIZED
7043 2.09 (2016-01-16) allow comments in PNM files
7044 16-bit-per-pixel TGA (not bit-per-component)
7045 info() for TGA could break due to .hdr handling
7046 info() for BMP to shares code instead of sloppy parse
7047 can use STBI_REALLOC_SIZED if allocator doesn't support realloc
7048 code cleanup
7049 2.08 (2015-09-13) fix to 2.07 cleanup, reading RGB PSD as RGBA
7050 2.07 (2015-09-13) fix compiler warnings
7051 partial animated GIF support
7052 limited 16-bpc PSD support
7053 #ifdef unused functions
7054 bug with < 92 byte PIC,PNM,HDR,TGA
7055 2.06 (2015-04-19) fix bug where PSD returns wrong '*comp' value
7056 2.05 (2015-04-19) fix bug in progressive JPEG handling, fix warning
7057 2.04 (2015-04-15) try to re-enable SIMD on MinGW 64-bit
7058 2.03 (2015-04-12) extra corruption checking (mmozeiko)
7059 stbi_set_flip_vertically_on_load (nguillemot)
7060 fix NEON support; fix mingw support
7061 2.02 (2015-01-19) fix incorrect assert, fix warning
7062 2.01 (2015-01-17) fix various warnings; suppress SIMD on gcc 32-bit without -msse2
7063 2.00b (2014-12-25) fix STBI_MALLOC in progressive JPEG
7064 2.00 (2014-12-25) optimize JPG, including x86 SSE2 & NEON SIMD (ryg)
7065 progressive JPEG (stb)
7066 PGM/PPM support (Ken Miller)
7067 STBI_MALLOC,STBI_REALLOC,STBI_FREE
7068 GIF bugfix -- seemingly never worked
7069 STBI_NO_*, STBI_ONLY_*
7070 1.48 (2014-12-14) fix incorrectly-named assert()
7071 1.47 (2014-12-14) 1/2/4-bit PNG support, both direct and paletted (Omar Cornut & stb)
7072 optimize PNG (ryg)
7073 fix bug in interlaced PNG with user-specified channel count (stb)
7074 1.46 (2014-08-26)
7075 fix broken tRNS chunk (colorkey-style transparency) in non-paletted PNG
7076 1.45 (2014-08-16)
7077 fix MSVC-ARM internal compiler error by wrapping malloc
7078 1.44 (2014-08-07)
7079 various warning fixes from Ronny Chevalier
7080 1.43 (2014-07-15)
7081 fix MSVC-only compiler problem in code changed in 1.42
7082 1.42 (2014-07-09)
7083 don't define _CRT_SECURE_NO_WARNINGS (affects user code)
7084 fixes to stbi__cleanup_jpeg path
7085 added STBI_ASSERT to avoid requiring assert.h
7086 1.41 (2014-06-25)
7087 fix search&replace from 1.36 that messed up comments/error messages
7088 1.40 (2014-06-22)
7089 fix gcc struct-initialization warning
7090 1.39 (2014-06-15)
7091 fix to TGA optimization when req_comp != number of components in TGA;
7092 fix to GIF loading because BMP wasn't rewinding (whoops, no GIFs in my test suite)
7093 add support for BMP version 5 (more ignored fields)
7094 1.38 (2014-06-06)
7095 suppress MSVC warnings on integer casts truncating values
7096 fix accidental rename of 'skip' field of I/O
7097 1.37 (2014-06-04)
7098 remove duplicate typedef
7099 1.36 (2014-06-03)
7100 convert to header file single-file library
7101 if de-iphone isn't set, load iphone images color-swapped instead of returning NULL
7102 1.35 (2014-05-27)
7103 various warnings
7104 fix broken STBI_SIMD path
7105 fix bug where stbi_load_from_file no longer left file pointer in correct place
7106 fix broken non-easy path for 32-bit BMP (possibly never used)
7107 TGA optimization by Arseny Kapoulkine
7108 1.34 (unknown)
7109 use STBI_NOTUSED in stbi__resample_row_generic(), fix one more leak in tga failure case
7110 1.33 (2011-07-14)
7111 make stbi_is_hdr work in STBI_NO_HDR (as specified), minor compiler-friendly improvements

```

```

7112 1.32 (2011-07-13)
7113 support for "info" function for all supported filetypes (SpartanJ)
7114 1.31 (2011-06-20)
7115 a few more leak fixes, bug in PNG handling (SpartanJ)
7116 1.30 (2011-06-11)
7117 added ability to load files via callbacks to accomidate custom input streams (Ben Wenger)
7118 removed deprecated format-specific test/load functions
7119 removed support for installable file formats (stbi_loader) -- would have been broken for IO callbacks
    anyway
7120 error cases in bmp and tga give messages and don't leak (Raymond Barbiero, grisha)
7121 fix inefficiency in decoding 32-bit BMP (David Woo)
7122 1.29 (2010-08-16)
7123 various warning fixes from Aurelien Pocheville
7124 1.28 (2010-08-01)
7125 fix bug in GIF palette transparency (SpartanJ)
7126 1.27 (2010-08-01)
7127 cast-to-stbi_uc to fix warnings
7128 1.26 (2010-07-24)
7129 fix bug in file buffering for PNG reported by SpartanJ
7130 1.25 (2010-07-17)
7131 refix trans_data warning (Won Chun)
7132 1.24 (2010-07-12)
7133 perf improvements reading from files on platforms with lock-heavy fgetc()
7134 minor perf improvements for jpeg
7135 deprecated type-specific functions so we'll get feedback if they're needed
7136 attempt to fix trans_data warning (Won Chun)
7137 1.23 fixed bug in iPhone support
7138 1.22 (2010-07-10)
7139 removed image *writing* support
7140 stbi_info support from Jetro Lauha
7141 GIF support from Jean-Marc Lienher
7142 iPhone PNG-extensions from James Brown
7143 warning-fixes from Nicolas Schulz and Janez Zemva (i.stbi_err. Janez (U+017D)emva)
7144 1.21 fix use of 'stbi_uc' in header (reported by jon blow)
7145 1.20 added support for Softimage PIC, by Tom Seddon
7146 1.19 bug in interlaced PNG corruption check (found by ryg)
7147 1.18 (2008-08-02)
7148 fix a threading bug (local mutable static)
7149 1.17 support interlaced PNG
7150 1.16 major bugfix - stbi_convert_format converted one too many pixels
7151 1.15 initialize some fields for thread safety
7152 1.14 fix threadsafe conversion bug
7153 header-file-only version (#define STBI_HEADER_FILE_ONLY before including)
7154 1.13 threadsafe
7155 1.12 const qualifiers in the API
7156 1.11 Support installable IDCT, colorspace conversion routines
7157 1.10 Fixes for 64-bit (don't use "unsigned long")
7158 optimized upsampling by Fabian "ryg" Giesen
7159 1.09 Fix format-conversion for PSD code (bad global variables!)
7160 1.08 Thatcher Ulrich's PSD code integrated by Nicolas Schulz
7161 1.07 attempt to fix C++ warning/errors again
7162 1.06 attempt to fix C++ warning/errors again
7163 1.05 fix TGA loading to return correct *comp and use good luminance calc
7164 1.04 default float alpha is 1, not 255; use 'void *' for stbi_image_free
7165 1.03 bugfixes to STBI_NO_STDIO, STBI_NO_HDR
7166 1.02 support for (subset of) HDR files, float interface for preferred access to them
7167 1.01 fix bug: possible bug in handling right-side up bmps... not sure
7168 fix bug: the stbi_bmp_load() and stbi_tga_load() functions didn't work at all
7169 1.00 interface to zlib that skips zlib header
7170 0.99 correct handling of alpha in palette
7171 0.98 TGA loader by lonesock; dynamically add loaders (untested)
7172 0.97 jpeg errors on too large a file; also catch another malloc failure
7173 0.96 fix detection of invalid v value - particleman@mollyrocket forum
7174 0.95 during header scan, seek to markers in case of padding
7175 0.94 STBI_NO_STDIO to disable stdio usage; rename all #defines the same
7176 0.93 handle jpegtran output; verbose errors
7177 0.92 read 4,8,16,24,32-bit BMP files of several formats
7178 0.91 output 24-bit Windows 3.0 BMP files
7179 0.90 fix a few more warnings; bump version number to approach 1.0
7180 0.61 bugfixes due to Marc LeBlanc, Christopher Lloyd
7181 0.60 fix compiling as c++
7182 0.59 fix warnings: merge Dave Moore's -Wall fixes
7183 0.58 fix bug: zlib uncompressed mode len/nlen was wrong endian
7184 0.57 fix bug: jpg last huffman symbol before marker was >9 bits but less than 16 available
7185 0.56 fix bug: zlib uncompressed mode len vs. nlen
7186 0.55 fix bug: restart_interval not initialized to 0
7187 0.54 allow NULL for 'int *comp'
7188 0.53 fix bug in png 3->4; speedup png decoding
7189 0.52 png handles req_comp=3,4 directly; minor cleanup; jpeg comments
7190 0.51 obey req_comp requests, 1-component jpegs return as 1-component,
7191 on 'test' only check type, not whether we support this variant
7192 0.50 (2006-11-19)
7193 first released version
7194 */

```


4.9 Texture.h

```

1 #pragma once
2 // GLEW
3 #include <GL/glew.h>
4
5 // Other Libs
6 #include "stb_image.h"
7
8 // Other includes
9 #include "Model.h"
10 #include <vector>
11
12
13 class TextureLoading
14 {
15 public:
16     static GLuint LoadTexture(GLchar *path)
17     {
18         unsigned int textureID;
19         glGenTextures(1, &textureID);
20
21         int width, height, nrComponents;
22         unsigned char *data = stbi_load(path, &width, &height, &nrComponents, 0);
23         if (data)
24         {
25             GLenum format;
26             if (nrComponents == 1)
27                 format = GL_RED;
28             else if (nrComponents == 3)
29                 format = GL_RGB;
30             else if (nrComponents == 4)
31                 format = GL_RGBA;
32
33             glBindTexture(GL_TEXTURE_2D, textureID);
34             glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
35             glGenerateMipmap(GL_TEXTURE_2D);
36
37             glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
38             glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
39             glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
40             glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
41
42             stbi_image_free(data);
43         }
44         else
45         {
46             std::cout << "Failed to load texture" << path << std::endl;
47             stbi_image_free(data);
48         }
49
50         return textureID;
51     }
52 }
53
54
55 static GLuint LoadCubemap(vector<const GLchar * > faces)
56 {
57     GLuint textureID;
58     glGenTextures(1, &textureID);
59
60     int width, height, nrChannels;
61     for (unsigned int i = 0; i < faces.size(); i++)
62     {
63         unsigned char *data = stbi_load(faces[i], &width, &height, &nrChannels, 0);
64         if (data)
65         {
66             glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height, 0, GL_RGB,
67                 GL_UNSIGNED_BYTE, data);
68             stbi_image_free(data);
69         }
70         else
71         {
72             std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;
73             stbi_image_free(data);
74         }
75     }
76     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
77     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
78     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
79     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
80     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
81     glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
82
83     return textureID;
84 }

```

```
85 };
```

Index

- [_frame](#), [5](#)
- [419048901_Proyecto_Gpo04.cpp](#), [11](#)
 - [main](#), [13](#)
 - [pointLightPositions](#), [13](#)
 - [spotLightDir](#), [13](#)
- [BoneMatrix](#), [5](#)
- [Camera](#), [6](#)
- [main](#)
 - [419048901_Proyecto_Gpo04.cpp](#), [13](#)
- [Mesh](#), [6](#)
- [MeshAnim](#), [6](#)
- [Model](#), [7](#)
- [ModelAnim](#), [7](#)
- [pointLightPositions](#)
 - [419048901_Proyecto_Gpo04.cpp](#), [13](#)
- [Shader](#), [8](#)
- [spotLightDir](#)
 - [419048901_Proyecto_Gpo04.cpp](#), [13](#)
- [stbi_io_callbacks](#), [8](#)
- [Texture](#), [8](#)
- [TextureLoading](#), [9](#)
- [Vertex](#), [9](#)
- [VertexBoneData](#), [9](#)