ProyectoFinal

Generated by Doxygen 1.9.4

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

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

Chapter 3

Class Documentation

3.1 _frame Struct Reference

Public Attributes

- float **posX**
- float **posY**
- float **posZ**
- float **incX**
- float **incY**
- float **incZ**
- float **rotRodIzq**
- 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 **MouseCallback** (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

```cpp
#pragma once

// Std. Includes
#include <vector>

// GL Includes
#define GLEW_STATIC
#include <GL/glew.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

// Defines several possible options for camera movement. Used as abstraction to stay away from
    window-system specific input methods
enum Camera_Movement
{
    FORWARD,
    BACKWARD,
    LEFT,
    RIGHT
};

// Default camera values
const GLfloat YAW = -90.0f;
const GLfloat PITCH = 0.0f;
const GLfloat SPEED = 10.0f;
const GLfloat SENSITIVTY = 0.25f;
const GLfloat ZOOM = 45.0f;

// An abstract camera class that processes input and calculates the corresponding Eular Angles, Vectors
    and Matrices for use in OpenGL
class Camera
{
public:
    // Constructor with vectors
    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(SENSITIVTY), zoom(ZOOM)
    {
        this->position = position;
        this->worldUp = up;
        this->yaw = yaw;
        this->pitch = pitch;
        this->updateCameraVectors();
    }

    // Constructor with scalar values
    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(SENSITIVTY), zoom(ZOOM)
    {
        this->position = glm::vec3(posX, posY, posZ);
        this->worldUp = glm::vec3(upX, upY, upZ);
        this->yaw = yaw;
        this->pitch = pitch;
        this->updateCameraVectors();
    }

    // Returns the view matrix calculated using Eular Angles and the LookAt Matrix
    glm::mat4 GetViewMatrix()
    {
        return glm::lookAt(this->position, this->position + this->front, this->up);
    }

    void Recorrido(GLfloat xOffset)//Modifica la rotaciecibiendo el ulo
    {
        this->yaw = xOffset;
        this->updateCameraVectors();
    }

    void MovimientoAutomatico(GLfloat velocidad) //Realiza un movimiento automatico hacia adelante
    {
        this->position += this->front * velocidad;
    }


    // 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)
    void ProcessKeyboard(Camera_Movement direction, GLfloat deltaTime)
    {
        GLfloat velocity = this->movementSpeed * deltaTime;

        if (direction == FORWARD)
        {
            this->position += this->front * velocity;
        }

        if (direction == BACKWARD)
        {
            this->position -= this->front * velocity;
        }

        if (direction == LEFT)
        {
            this->position -= this->right * velocity;
        }

        if (direction == RIGHT)
        {
            this->position += this->right * velocity;
        }
    }

    // Processes input received from a mouse input system. Expects the offset value in both the x and y
    direction.
    void ProcessMouseMovement(GLfloat xOffset, GLfloat yOffset, GLboolean constrainPitch = true)
    {
         xOffset *= this->mouseSensitivity;
         yOffset *= this->mouseSensitivity;

         this->yaw += xOffset;
         this->pitch += yOffset;

         // Make sure that when pitch is out of bounds, screen doesn't get flipped
         if (constrainPitch)
         {
             if (this->pitch > 100.0f)
             {
                 this->pitch = 89.0f;
             }

             if (this->pitch < -89.0f)
             {
                 this->pitch = -89.0f;
             }
         }

         // Update Front, Right and Up Vectors using the updated Eular angles
         this->updateCameraVectors();
     }


     // Processes input received from a mouse scroll-wheel event. Only requires input on the vertical
    wheel-axis
     void ProcessMouseScroll(GLfloat yOffset)
     {

     }

     GLfloat GetZoom()
     {
         return this->zoom;
     }

     glm::vec3 GetPosition()
     {
         return this->position;
     }

     glm::vec3 GetFront()
     {
         return this->front;
     }

private:
     // Camera Attributes
     glm::vec3 position;
     glm::vec3 front;
     glm::vec3 up;
     glm::vec3 right;
     glm::vec3 worldUp;

     // Eular Angles
     GLfloat yaw;
     GLfloat pitch;

     // Camera options
     GLfloat movementSpeed;
     GLfloat mouseSensitivity;
     GLfloat zoom;

     // Calculates the front vector from the Camera's (updated) Eular Angles
     void updateCameraVectors()
     {
         // Calculate the new Front vector
         glm::vec3 front;
         front.x = cos(glm::radians(this->yaw)) * cos(glm::radians(this->pitch));
         front.y = sin(glm::radians(this->pitch));
         front.z = sin(glm::radians(this->yaw)) * cos(glm::radians(this->pitch));
         this->front = glm::normalize(front);
         // Also re-calculate the Right and Up vector
         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.
         this->up = glm::normalize(glm::cross(this->right, this->front));
     }
 };
```

4.3 Mesh.h

```
#ifndef MESH_H
#define MESH_H

//#include "glad.h"// holds all OpenGL type declarations

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include "shader.h"

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
#include <vector>
using namespace std;

struct Vertex {
    // position
    glm::vec3 Position;
    // normal
    glm::vec3 Normal;
    // texCoords
    glm::vec2 TexCoords;
    // tangent
    glm::vec3 Tangent;
    // bitangent
    glm::vec3 Bitangent;
};

struct Texture {
    unsigned int id;
    string type;
    string path;
};

class Mesh {
public:
    /*  Mesh Data  */
    vector<Vertex> vertices;
    vector<unsigned int> indices;
    vector<Texture> textures;
    unsigned int VAO;

    /*  Functions  */
    // constructor
    Mesh(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures)
    {
        this->vertices = vertices;
        this->indices = indices;
        this->textures = textures;

        // now that we have all the required data, set the vertex buffers and its attribute pointers.
        setupMesh();
    }

    // render the mesh
    void Draw(Shader shader)
    {
        // bind appropriate textures
        unsigned int diffuseNr  = 1;
        unsigned int specularNr = 1;
        unsigned int normalNr   = 1;
        unsigned int heightNr   = 1;
        for(unsigned int i = 0; i < textures.size(); i++)
        {
            glActiveTexture(GL_TEXTURE0 + i); // active proper texture unit before binding
            // retrieve texture number (the N in diffuse_textureN)
            string number;
            string name = textures[i].type;
            if(name == "texture_diffuse")
                number = std::to_string(diffuseNr++);
            else if(name == "texture_specular")
                number = std::to_string(specularNr++); // transfer unsigned int to stream
            else if(name == "texture_normal")
                number = std::to_string(normalNr++); // transfer unsigned int to stream
             else if(name == "texture_height")
                number = std::to_string(heightNr++); // transfer unsigned int to stream

            // now set the sampler to the correct texture unit
            glUniform1i(glGetUniformLocation(shader.Program, (name + number).c_str()), i);    // AQUI ES DONDE SE ASIGNAN LOS UNIFORM A LOS SHADERS AHHHHHHHHHHHH
            // and finally bind the texture
            glBindTexture(GL_TEXTURE_2D, textures[i].id);
        }

        // draw mesh
        glBindVertexArray(VAO);
        glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
        glBindVertexArray(0);

        // always good practice to set everything back to defaults once configured.
        glActiveTexture(GL_TEXTURE0);
    }

private:
    /*  Render data  */
    unsigned int VBO, EBO;

    /*  Functions    */
     // initializes all the buffer objects/arrays
     void setupMesh()
     {
         // create buffers/arrays
         glGenVertexArrays(1, &VAO);
         glGenBuffers(1, &VBO);
         glGenBuffers(1, &EBO);

         glBindVertexArray(VAO);
         // load data into vertex buffers
         glBindBuffer(GL_ARRAY_BUFFER, VBO);
         // A great thing about structs is that their memory layout is sequential for all its items.
         // The effect is that we can simply pass a pointer to the struct and it translates perfectly to a glm::vec3/2 array which
         // again translates to 3/2 floats which translates to a byte array.
         glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);

         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
         glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);

         // set the vertex attribute pointers
         // vertex Positions
         glEnableVertexAttribArray(0);
         glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
         // vertex normals
         glEnableVertexAttribArray(1);
         glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
         // vertex texture coords
         glEnableVertexAttribArray(2);
         glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));
         // vertex tangent
         glEnableVertexAttribArray(3);
         glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Tangent));
         // vertex bitangent
         glEnableVertexAttribArray(4);
         glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Bitangent));

         glBindVertexArray(0);
     }
 };
 #endif

```

4.4 meshAnim.h

```
#ifndef MESH_ANIM_H
#define MESH_ANIM_H

//#include <glad.h>// holds all OpenGL type declarations

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include "shader.h"
#include "mesh.h"

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
#include <vector>
using namespace std;

typedef unsigned int uint;
#define NUM_BONES_PER_VEREX 4

struct BoneMatrix
{
    aiMatrix4x4 offset_matrix;
    aiMatrix4x4 final_world_transform;
};

struct VertexBoneData
{
    uint ids[NUM_BONES_PER_VEREX];   // we have 4 bone ids for EACH vertex & 4 weights for EACH vertex
    float weights[NUM_BONES_PER_VEREX];

    VertexBoneData()
    {
        memset(ids, 0, sizeof(ids));    // init all values in array = 0
        memset(weights, 0, sizeof(weights));
    }

    void addBoneData(uint bone_id, float weight)
    {
        for (uint i = 0; i < NUM_BONES_PER_VEREX; i++)
        {
            if (weights[i] == 0.0)
            {
                ids[i] = bone_id;
                weights[i] = weight;
                return;
            }
        }
    }
};

class MeshAnim {
public:
    /*  Mesh Data  */
    vector<Vertex> vertices;
    vector<unsigned int> indices;
    vector<Texture> textures;
    vector<VertexBoneData> bones_id_weights_for_each_vertex;
    unsigned int VAO;

    /*  Functions  */
    // constructor
    MeshAnim(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures)
    {
        this->vertices = vertices;
        this->indices = indices;
        this->textures = textures;

        // now that we have all the required data, set the vertex buffers and its attribute pointers.
        setupMesh();
    }

    MeshAnim(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures,
    vector<VertexBoneData> bone_id_weights)
    {
        this->vertices = vertices;
        this->indices = indices;
        this->textures = textures;
        bones_id_weights_for_each_vertex = bone_id_weights;

        // now that we have all the required data, set the vertex buffers and its attribute pointers.
        setupMesh();
    }

```

```
   // render the mesh
   void Draw(Shader shader)
   {
       // bind appropriate textures
       unsigned int diffuseNr  = 1;
       unsigned int specularNr = 1;
       unsigned int normalNr   = 1;
       unsigned int heightNr   = 1;
       for(unsigned int i = 0; i < textures.size(); i++)
       {
           glActiveTexture(GL_TEXTURE0 + i); // active proper texture unit before binding
           // retrieve texture number (the N in diffuse_textureN)
           string number;
           string name = textures[i].type;
           if(name == "texture_diffuse")
                number = std::to_string(diffuseNr++);
            else if(name == "texture_specular")
                number = std::to_string(specularNr++); // transfer unsigned int to stream
            else if(name == "texture_normal")
                number = std::to_string(normalNr++); // transfer unsigned int to stream
             else if(name == "texture_height")
                number = std::to_string(heightNr++); // transfer unsigned int to stream

            // now set the sampler to the correct texture unit
            glUniform1i(glGetUniformLocation(shader.Program, (name + number).c_str()), i);    // AQUI ES DONDE SE ASIGNAN LOS UNIFORM A LOS SHADERS AHHHHHHHHHHHH
            // and finally bind the texture
            glBindTexture(GL_TEXTURE_2D, textures[i].id);
        }

        // draw mesh
        glBindVertexArray(VAO);
        glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
        glBindVertexArray(0);

        // always good practice to set everything back to defaults once configured.
        glActiveTexture(GL_TEXTURE0);
    }

private:
    /*  Render data  */
    unsigned int VBO, EBO, VBO_bones;

    /*  Functions    */
    // initializes all the buffer objects/arrays
    void setupMesh()
    {
        // create buffers/arrays
        glGenVertexArrays(1, &VAO);
        glGenBuffers(1, &VBO);
        glGenBuffers(1, &EBO);
        glGenBuffers(1, &VBO_bones);

        glBindVertexArray(VAO);
        // load data into vertex buffers
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        // A great thing about structs is that their memory layout is sequential for all its items.
        // The effect is that we can simply pass a pointer to the struct and it translates perfectly to a glm::vec3/2 array which
        // again translates to 3/2 floats which translates to a byte array.
        glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);

        // bones
        glBindBuffer(GL_ARRAY_BUFFER, VBO_bones);
        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);

        // indices
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);

        // Se liga primero el buffer de los vertices
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        // set the vertex attribute pointers
        // vertex Positions
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
        // vertex normals
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
        // vertex texture coords
        glEnableVertexAttribArray(2);
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));
        // vertex tangent
        glEnableVertexAttribArray(3);
        glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
    Tangent));
        // vertex bitangent
        glEnableVertexAttribArray(4);
        glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
    Bitangent));

        // Se liga el buffer de los bones
        glBindBuffer(GL_ARRAY_BUFFER, VBO_bones);
        // set the bones atrribute pointers
        glEnableVertexAttribArray(5);
        glVertexAttribIPointer(5, 4, GL_INT, sizeof(VertexBoneData), (void*)0);
        glEnableVertexAttribArray(6);
        glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(VertexBoneData),
    (void*)offsetof(VertexBoneData, weights));

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBindVertexArray(0);
    }
};
#endif

```

4.5 Model.h

```
#ifndef MODEL_H
#define MODEL_H

//#include "glad.h"

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>

#include "mesh.h"
#include "shader.h"

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
#include <map>
#include <vector>
using namespace std;

unsigned int TextureFromFile(const char *path, const string &directory, bool gamma = false);

class Model
{
public:
    /*  Model Data */
    vector<Texture> textures_loaded;    // stores all the textures loaded so far, optimization to make
    sure textures aren't loaded more than once.
    vector<Mesh> meshes;
    string directory;
    bool gammaCorrection;

    /*  Functions   */
    // constructor, expects a filepath to a 3D model.
    Model(string const &path, bool gamma = false) : gammaCorrection(gamma)
    {
        loadModel(path);
    }

    // draws the model, and thus all its meshes
    void Draw(Shader shader)
    {
        for(unsigned int i = 0; i < meshes.size(); i++)
            meshes[i].Draw(shader);
    }

private:
    /*  Functions   */
    // loads a model with supported ASSIMP extensions from file and stores the resulting meshes in the
    meshes vector.
    void loadModel(string const &path)
    {
        // read file via ASSIMP
        Assimp::Importer importer;
        const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs | aiProcess_CalcTangentSpace);
        // check for errors
        if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) // if is Not Zero
        {
            cout « "ERROR::ASSIMP:: " « importer.GetErrorString() « endl;
            return;
        }
        // retrieve the directory path of the filepath
        directory = path.substr(0, path.find_last_of('/'));

        // process ASSIMP's root node recursively
        processNode(scene->mRootNode, scene);
    }

    // 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).
    void processNode(aiNode *node, const aiScene *scene)
    {
        // process each mesh located at the current node
        for(unsigned int i = 0; i < node->mNumMeshes; i++)
        {
            // the node object only contains indices to index the actual objects in the scene.
            // the scene contains all the data, node is just to keep stuff organized (like relations between nodes).
            aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
            meshes.push_back(processMesh(mesh, scene));
        }
        // after we've processed all of the meshes (if any) we then recursively process each of the children nodes
        for(unsigned int i = 0; i < node->mNumChildren; i++)
        {
            processNode(node->mChildren[i], scene);
        }

    }

    Mesh processMesh(aiMesh *mesh, const aiScene *scene)
    {
        // data to fill
        vector<Vertex> vertices;
        vector<unsigned int> indices;
        vector<Texture> textures;

        // Walk through each of the mesh's vertices
        for(unsigned int i = 0; i < mesh->mNumVertices; i++)
        {
            Vertex vertex;
             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.
             // positions
             vector.x = mesh->mVertices[i].x;
             vector.y = mesh->mVertices[i].y;
             vector.z = mesh->mVertices[i].z;
             vertex.Position = vector;
             // normals
             vector.x = mesh->mNormals[i].x;
             vector.y = mesh->mNormals[i].y;
             vector.z = mesh->mNormals[i].z;
             vertex.Normal = vector;
             // texture coordinates
             if(mesh->mTextureCoords[0]) // does the mesh contain texture coordinates?
             {
                 glm::vec2 vec;
                 // a vertex can contain up to 8 different texture coordinates. We thus make the assumption that we won't
                 // use models where a vertex can have multiple texture coordinates so we always take the first set (0).
                 vec.x = mesh->mTextureCoords[0][i].x;
                 vec.y = mesh->mTextureCoords[0][i].y;
                 vertex.TexCoords = vec;
             }
             else
                 vertex.TexCoords = glm::vec2(0.0f, 0.0f);
             // tangent
      /*       vector.x = mesh->mTangents[i].x;
             vector.y = mesh->mTangents[i].y;
             vector.z = mesh->mTangents[i].z;
             vertex.Tangent = vector;*/
             // bitangent
            /* vector.x = mesh->mBitangents[i].x;
             vector.y = mesh->mBitangents[i].y;
             vector.z = mesh->mBitangents[i].z;*/
             vertex.Bitangent = vector;
             vertices.push_back(vertex);
         }
        // now wak through each of the mesh's faces (a face is a mesh its triangle) and retrieve the corresponding vertex indices.
        for(unsigned int i = 0; i < mesh->mNumFaces; i++)
        {
            aiFace face = mesh->mFaces[i];
            // retrieve all indices of the face and store them in the indices vector
            for(unsigned int j = 0; j < face.mNumIndices; j++)
                indices.push_back(face.mIndices[j]);
        }
        // process materials
        aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
        // we assume a convention for sampler names in the shaders. Each diffuse texture should be named
        // as 'texture_diffuseN' where N is a sequential number ranging from 1 to MAX_SAMPLER_NUMBER.
        // Same applies to other texture as the following list summarizes:
        // diffuse: texture_diffuseN
        // specular: texture_specularN
        // normal: texture_normalN

        // 1. diffuse maps
        vector<Texture> diffuseMaps = loadMaterialTextures(material, aiTextureType_DIFFUSE, "texture_diffuse");
        textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());
        // 2. specular maps
        vector<Texture> specularMaps = loadMaterialTextures(material, aiTextureType_SPECULAR, "texture_specular");
        textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());
        // 3. normal maps
        std::vector<Texture> normalMaps = loadMaterialTextures(material, aiTextureType_HEIGHT, "texture_normal");
        textures.insert(textures.end(), normalMaps.begin(), normalMaps.end());
        // 4. height maps
        std::vector<Texture> heightMaps = loadMaterialTextures(material, aiTextureType_AMBIENT, "texture_height");
        textures.insert(textures.end(), heightMaps.begin(), heightMaps.end());

        // return a mesh object created from the extracted mesh data
        return Mesh(vertices, indices, textures);
    }

    // checks all material textures of a given type and loads the textures if they're not loaded yet.
    // the required info is returned as a Texture struct.
    vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type, string typeName)
    {
        vector<Texture> textures;
        for(unsigned int i = 0; i < mat->GetTextureCount(type); i++)
        {
            aiString str;
            mat->GetTexture(type, i, &str);
            // check if texture was loaded before and if so, continue to next iteration: skip loading a new texture
            bool skip = false;
            for(unsigned int j = 0; j < textures_loaded.size(); j++)
            {
                if(std::strcmp(textures_loaded[j].path.data(), str.C_Str()) == 0)
                {
                    textures.push_back(textures_loaded[j]);
                    skip = true; // a texture with the same filepath has already been loaded, continue to next one. (optimization)
                    break;
                }
            }
            if(!skip)
            {   // if texture hasn't been loaded already, load it
                Texture texture;
                texture.id = TextureFromFile(str.C_Str(), this->directory);
                texture.type = typeName;
                texture.path = str.C_Str();
                textures.push_back(texture);
                textures_loaded.push_back(texture);  // store it as texture loaded for entire model, to ensure we won't unnecesery load duplicate textures.
            }
        }
        return textures;
    }
};


unsigned int TextureFromFile(const char *path, const string &directory, bool gamma)
{
    string filename = string(path);
    filename = directory + '/' + filename;

    unsigned int textureID;
    glGenTextures(1, &textureID);

    int width, height, nrComponents;
    unsigned char *data = stbi_load(filename.c_str(), &width, &height, &nrComponents, 0);
    if (data)
    {
        GLenum format;
        if (nrComponents == 1)
            format = GL_RED;
        else if (nrComponents == 3)
            format = GL_RGB;
        else if (nrComponents == 4)
            format = GL_RGBA;

        glBindTexture(GL_TEXTURE_2D, textureID);
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        stbi_image_free(data);
    }
    else
    {
        std::cout « "Texture failed to load at path: " « path « std::endl;
        stbi_image_free(data);
    }

    return textureID;
}
#endif
```

4.6 modelAnim.h

```
#ifndef MODEL_ANIM_H
#define MODEL_ANIM_H

//#include "glad.h"

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>

#include <GLFW/glfw3.h>
#include "meshAnim.h"
#include "model.h"
#include "shader.h"
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
#include <map>
#include <vector>
using namespace std;

class ModelAnim
{
public:
    /*  Model Data */
    vector<Texture> textures_loaded;    // stores all the textures loaded so far, optimization to make
    sure textures aren't loaded more than once.
    vector<MeshAnim> meshes;
    string directory;
    bool gammaCorrection;

    /* Importacion base */
    Assimp::Importer importer;
    const aiScene* scene;

    /* Huesos */
    static const uint MAX_BONES = 100;

    map<string, uint> m_bone_mapping; // maps a bone name and their index
    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;

    /*  Functions   */
    // constructor, expects a filepath to a 3D model.
    ModelAnim(string const &path, bool gamma = false) : gammaCorrection(gamma)
    {
        loadModel(path);
    }

    void initShaders(GLuint shader_program)
    {
        for (uint i = 0; i < MAX_BONES; i++) // get location all matrices of bones
        {
            string name = "bones[" + to_string(i) + "]";// name like in shader
            m_bone_location[i] = glGetUniformLocation(shader_program, name.c_str());
        }

        // rotate head AND AXIS(y_z) about x !!!!!  Not be gimbal lock
        //rotate_head_xz *= glm::quat(cos(glm::radians(-45.0f / 2)), sin(glm::radians(-45.0f / 2)) * glm::vec3(1.0f, 0.0f, 0.0f));
    }

    // draws the model, and thus all its meshes
    void Draw(Shader shader)
    {
        // Calculo de las animaciones
        vector<aiMatrix4x4> transforms;
        boneTransform((double)glfwGetTime(), transforms);

        for (uint i = 0; i < transforms.size(); i++) // move all matrices for actual model position to shader
        {
            glUniformMatrix4fv(m_bone_location[i], 1, GL_TRUE, (const GLfloat*)&transforms[i]);
        }

        for(unsigned int i = 0; i < meshes.size(); i++)
            meshes[i].Draw(shader);
    }

private:

    /*  Functions   */
    // loads a model with supported ASSIMP extensions from file and stores the resulting meshes in the meshes vector.
    void loadModel(string const &path)
    {
        // read file via ASSIMP
        scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs | aiProcess_CalcTangentSpace);
        // check for errors
        if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) // if is Not Zero
        {
            cout « "ERROR::ASSIMP:: " « importer.GetErrorString() « endl;
            return;
        }

        m_global_inverse_transform = scene->mRootNode->mTransformation;
         m_global_inverse_transform.Inverse();

         if (scene->mAnimations[0]->mTicksPerSecond != 0.0)
         {
             ticks_per_second = scene->mAnimations[0]->mTicksPerSecond;
         }
         else
         {
             ticks_per_second = 25.0f;
         }

         // retrieve the directory path of the filepath
         directory = path.substr(0, path.find_last_of('/'));

         cout « "scene->HasAnimations() 1: " « scene->HasAnimations() « endl;
         cout « "scene->mNumMeshes 1: " « scene->mNumMeshes « endl;
         cout « "scene->mAnimations[0]->mNumChannels 1: " « scene->mAnimations[0]->mNumChannels « endl;
         cout « "scene->mAnimations[0]->mDuration 1: " « scene->mAnimations[0]->mDuration « endl;
         cout « "scene->mAnimations[0]->mTicksPerSecond 1: " « scene->mAnimations[0]->mTicksPerSecond « endl « endl;

         cout « "       name nodes : " « endl;
         showNodeName(scene->mRootNode);
         cout « endl;

         cout « "       name bones : " « endl;
         //processNode(scene->mRootNode, scene);
         // process ASSIMP's root node recursively
         processNode(scene->mRootNode, scene);

         cout « "       name nodes animation : " « endl;
         for (uint i = 0; i < scene->mAnimations[0]->mNumChannels; i++)
         {
             cout « scene->mAnimations[0]->mChannels[i]->mNodeName.C_Str() « endl;
         }
         cout « endl;
     }

     void showNodeName(aiNode* node)
     {
         cout « node->mName.data « endl;
         for (uint i = 0; i < node->mNumChildren; i++)
         {
             showNodeName(node->mChildren[i]);
         }
     }

     // 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).
     void processNode(aiNode *node, const aiScene *scene)
     {
         // process each mesh located at the current node
         for(unsigned int i = 0; i < node->mNumMeshes; i++)
         {
             // the node object only contains indices to index the actual objects in the scene.
             // the scene contains all the data, node is just to keep stuff organized (like relations
    between nodes).
             aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
             meshes.push_back(processMesh(mesh, scene));
         }
         // after we've processed all of the meshes (if any) we then recursively process each of the
    children nodes
         for(unsigned int i = 0; i < node->mNumChildren; i++)
         {
             processNode(node->mChildren[i], scene);
         }

     }

     MeshAnim processMesh(aiMesh *mesh, const aiScene *scene)
     {
         std::cout « "bones: " « mesh->mNumBones « " vertices: " « mesh->mNumVertices « std::endl;
         // data to fill
         vector<Vertex> vertices;
         vector<unsigned int> indices;
         vector<Texture> textures;
         vector<VertexBoneData> bones_id_weights_for_each_vertex;


         //Tal vez haya que hacer resize de los vectores
         vertices.reserve(mesh->mNumVertices);
         indices.reserve(mesh->mNumVertices);
         bones_id_weights_for_each_vertex.resize(mesh->mNumVertices);


         // Walk through each of the mesh's vertices
         for(unsigned int i = 0; i < mesh->mNumVertices; i++)
         {
             Vertex vertex;
             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.
             // positions
             vector.x = mesh->mVertices[i].x;
             vector.y = mesh->mVertices[i].y;
             vector.z = mesh->mVertices[i].z;
             vertex.Position = vector;
             // normals
             vector.x = mesh->mNormals[i].x;
             vector.y = mesh->mNormals[i].y;
             vector.z = mesh->mNormals[i].z;
             vertex.Normal = vector;
             // texture coordinates
             if(mesh->mTextureCoords[0]) // does the mesh contain texture coordinates?
             {
                 glm::vec2 vec;
                 // a vertex can contain up to 8 different texture coordinates. We thus make the
    assumption that we won't
                 // use models where a vertex can have multiple texture coordinates so we always take the
    first set (0).
                 vec.x = mesh->mTextureCoords[0][i].x;
                 vec.y = mesh->mTextureCoords[0][i].y;
                 vertex.TexCoords = vec;
             }
             else
                 vertex.TexCoords = glm::vec2(0.0f, 0.0f);
             // tangent
             vector.x = mesh->mTangents[i].x;
             vector.y = mesh->mTangents[i].y;
             vector.z = mesh->mTangents[i].z;
             vertex.Tangent = vector;
             // bitangent
             vector.x = mesh->mBitangents[i].x;
             vector.y = mesh->mBitangents[i].y;
             vector.z = mesh->mBitangents[i].z;
             vertex.Bitangent = vector;
             vertices.push_back(vertex);
         }
         // now wak through each of the mesh’s faces (a face is a mesh its triangle) and retrieve the corresponding vertex indices.
         for(unsigned int i = 0; i < mesh->mNumFaces; i++)
         {
             aiFace face = mesh->mFaces[i];
             // retrieve all indices of the face and store them in the indices vector
             for(unsigned int j = 0; j < face.mNumIndices; j++)
                 indices.push_back(face.mIndices[j]);
         }
         // process materials
         aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
         // we assume a convention for sampler names in the shaders. Each diffuse texture should be named
         // as ’texture_diffuseN’ where N is a sequential number ranging from 1 to MAX_SAMPLER_NUMBER.
         // Same applies to other texture as the following list summarizes:
         // diffuse: texture_diffuseN
         // specular: texture_specularN
         // normal: texture_normalN

         // 1. diffuse maps
         vector<Texture> diffuseMaps = loadMaterialTextures(material, aiTextureType_DIFFUSE, "texture_diffuse");
         textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());
         // 2. specular maps
         vector<Texture> specularMaps = loadMaterialTextures(material, aiTextureType_SPECULAR, "texture_specular");
         textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());
         // 3. normal maps
         std::vector<Texture> normalMaps = loadMaterialTextures(material, aiTextureType_HEIGHT, "texture_normal");
         textures.insert(textures.end(), normalMaps.begin(), normalMaps.end());
         // 4. height maps
         std::vector<Texture> heightMaps = loadMaterialTextures(material, aiTextureType_AMBIENT, "texture_height");
         textures.insert(textures.end(), heightMaps.begin(), heightMaps.end());

         // load bones
         for (uint i = 0; i < mesh->mNumBones; i++)
         {
             uint bone_index = 0;
             string bone_name(mesh->mBones[i]->mName.data);

             // Impresión de los nombres de los huesos
             cout « mesh->mBones[i]->mName.data « endl;

             if (m_bone_mapping.find(bone_name) == m_bone_mapping.end()) // ��������� ��� �� � ������� ��������
             {
                 // Allocate an index for a new bone
                 bone_index = m_num_bones;
                 m_num_bones++;
                 BoneMatrix bi;
                 m_bone_matrices.push_back(bi);

                 m_bone_matrices[bone_index].offset_matrix = mesh->mBones[i]->mOffsetMatrix; // Aqui hay que agregar el giro HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
                 m_bone_mapping[bone_name] = bone_index;

                 //cout « "bone_name: " « bone_name « "            bone_index: " « bone_index « endl;
             }
             else
             {
                 bone_index = m_bone_mapping[bone_name];
             }

             for (uint j = 0; j < mesh->mBones[i]->mNumWeights; j++)
             {
                 uint vertex_id = mesh->mBones[i]->mWeights[j].mVertexId; // �� ������� �� ������ ����� �������� �����
                 float weight = mesh->mBones[i]->mWeights[j].mWeight;
                 bones_id_weights_for_each_vertex[vertex_id].addBoneData(bone_index, weight); // � ������ ������� ����� ����� � �� ���

                 // ������ ������� vertex_id �� ������ ����� � �������� bone_index  ����� ��� weight
                 //cout « " vertex_id: " « vertex_id « "  bone_index: " « bone_index « "        weight: " « weight « endl;
             }
         }

```

```
        // return a mesh object created from the extracted mesh data
        return MeshAnim(vertices, indices, textures, bones_id_weights_for_each_vertex);
    }

    // checks all material textures of a given type and loads the textures if they're not loaded yet.
    // the required info is returned as a Texture struct.
    vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type, string typeName)
    {
        vector<Texture> textures;
        for(unsigned int i = 0; i < mat->GetTextureCount(type); i++)
        {
            aiString str;
            mat->GetTexture(type, i, &str);
            // check if texture was loaded before and if so, continue to next iteration: skip loading a new texture
            bool skip = false;
            for(unsigned int j = 0; j < textures_loaded.size(); j++)
            {
                if(std::strcmp(textures_loaded[j].path.data(), str.C_Str()) == 0)
                {
                    textures.push_back(textures_loaded[j]);
                    skip = true; // a texture with the same filepath has already been loaded, continue to next one. (optimization)
                    break;
                }
            }
            if(!skip)
            {   // if texture hasn't been loaded already, load it
                Texture texture;
                texture.id = TextureFromFile(str.C_Str(), this->directory);
                texture.type = typeName;
                texture.path = str.C_Str();
                textures.push_back(texture);
                textures_loaded.push_back(texture);  // store it as texture loaded for entire model, to ensure we won't unnecesery load duplicate textures.
            }
        }
        return textures;
    }

    uint findPosition(float p_animation_time, const aiNodeAnim* p_node_anim)
    {
        // ����� ���� ������� ����� ����� ����� ������� ���������� ����� ������ ��������
        for (uint i = 0; i < p_node_anim->mNumPositionKeys - 1; i++) // �������� ����� ��������
        {
            if (p_animation_time < (float)p_node_anim->mPositionKeys[i + 1].mTime) // �������� �� �������� ��������� !!!
            {
                return i; // �� ������� ������ �������� !!!!!!!!!!!!!!!!!! ����������������������������
            }
        }

        assert(0);
        return 0;
    }

    uint findRotation(float p_animation_time, const aiNodeAnim* p_node_anim)
    {
        // ����� ���� ������� ����� ����� ����� ������� ���������� ����� ������ ��������
        for (uint i = 0; i < p_node_anim->mNumRotationKeys - 1; i++) // �������� ����� ��������
        {
            if (p_animation_time < (float)p_node_anim->mRotationKeys[i + 1].mTime) // �������� �� �������� ��������� !!!
            {
                return i; // �� ������� ������ �������� !!!!!!!!!!!!!!!!!! ����������������������������
            }
        }

        assert(0);
        return 0;
    }

    uint findScaling(float p_animation_time, const aiNodeAnim* p_node_anim)
    {
        // ����� ���� ������� ����� ����� ����� ������� ���������� ����� ������ ��������
        for (uint i = 0; i < p_node_anim->mNumScalingKeys - 1; i++) // �������� ����� ��������
        {
            if (p_animation_time < (float)p_node_anim->mScalingKeys[i + 1].mTime) // �������� �� �������� ��������� !!!
            {
                return i; // �� ������� ������ �������� !!!!!!!!!!!!!!!!!! ����������������������������
            }
        }

        assert(0);
        return 0;
    }
```

```

 aiVector3D calcInterpolatedPosition(float p_animation_time, const aiNodeAnim* p_node_anim)
 {
     if (p_node_anim->mNumPositionKeys == 1) // Keys ��� ������� �����
     {
         return p_node_anim->mPositionKeys[0].mValue;
     }

     uint position_index = findPosition(p_animation_time, p_node_anim); // ������ ������ �������� ����� ������� ������
     uint next_position_index = position_index + 1; // ������ ��������� �������� �����
     assert(next_position_index < p_node_anim->mNumPositionKeys);
     // ���� ����� �������
     float delta_time = (float)(p_node_anim->mPositionKeys[next_position_index].mTime - p_node_anim->mPositionKeys[position_index].mTime);
     // ������ = (���� ������� ������ �� ������ �������� ��������� �����) / �� ���� ����� �������
     float factor = (p_animation_time - (float)p_node_anim->mPositionKeys[position_index].mTime) / delta_time;
     assert(factor >= 0.0f && factor <= 1.0f);
     aiVector3D start = p_node_anim->mPositionKeys[position_index].mValue;
     aiVector3D end = p_node_anim->mPositionKeys[next_position_index].mValue;
     aiVector3D delta = end - start;

     return start + factor * delta;
 }

 aiQuaternion calcInterpolatedRotation(float p_animation_time, const aiNodeAnim* p_node_anim)
 {
     if (p_node_anim->mNumRotationKeys == 1) // Keys ��� ������� �����
     {
         return p_node_anim->mRotationKeys[0].mValue;
     }

     uint rotation_index = findRotation(p_animation_time, p_node_anim); // ������ ������ �������� ����� ������� ������
     uint next_rotation_index = rotation_index + 1; // ������ ��������� �������� �����
     assert(next_rotation_index < p_node_anim->mNumRotationKeys);
     // ���� ����� �������
     float delta_time = (float)(p_node_anim->mRotationKeys[next_rotation_index].mTime - p_node_anim->mRotationKeys[rotation_index].mTime);
     // ������ = (���� ������� ������ �� ������ �������� ��������� �����) / �� ���� ����� �������
     float factor = (p_animation_time - (float)p_node_anim->mRotationKeys[rotation_index].mTime) / delta_time;

     //cout « "p_node_anim->mRotationKeys[rotation_index].mTime: " « p_node_anim->mRotationKeys[rotation_index].mTime « endl;
     //cout « "p_node_anim->mRotationKeys[next_rotaion_index].mTime: " « p_node_anim->mRotationKeys[next_rotation_index].mTime « endl;
     //cout « "delta_time: " « delta_time « endl;
     //cout « "animation_time: " « p_animation_time « endl;
     //cout « "animation_time - mRotationKeys[rotation_index].mTime: " « (p_animation_time - (float)p_node_anim->mRotationKeys[rotation_index].mTime) « endl;
     //cout « "factor: " « factor « endl « endl « endl;

     assert(factor >= 0.0f && factor <= 1.0f);
     aiQuaternion start_quat = p_node_anim->mRotationKeys[rotation_index].mValue;
     aiQuaternion end_quat = p_node_anim->mRotationKeys[next_rotation_index].mValue;

     return nlerp(start_quat, end_quat, factor);
 }

 aiVector3D calcInterpolatedScaling(float p_animation_time, const aiNodeAnim* p_node_anim)
 {
     if (p_node_anim->mNumScalingKeys == 1) // Keys ��� ������� �����
     {
         return p_node_anim->mScalingKeys[0].mValue;
     }

     uint scaling_index = findScaling(p_animation_time, p_node_anim); // ������ ������ �������� ����� ������� ������
     uint next_scaling_index = scaling_index + 1; // ������ ��������� �������� �����
     assert(next_scaling_index < p_node_anim->mNumScalingKeys);
     // ���� ����� �������
     float delta_time = (float)(p_node_anim->mScalingKeys[next_scaling_index].mTime - p_node_anim->mScalingKeys[scaling_index].mTime);
     // ������ = (���� ������� ������ �� ������ �������� ��������� �����) / �� ���� ����� �������
     float  factor = (p_animation_time - (float)p_node_anim->mScalingKeys[scaling_index].mTime) / delta_time;
     //cout « "p_animation_time: " « p_animation_time « " " « "mTime: " « (float)p_node_anim->mScalingKeys[scaling_index].mTime « endl « endl « endl;
     assert(factor >= 0.0f && factor <= 1.0f);
     aiVector3D start = p_node_anim->mScalingKeys[scaling_index].mValue;
     aiVector3D end = p_node_anim->mScalingKeys[next_scaling_index].mValue;
     aiVector3D delta = end - start;

     return start + factor * delta;
 }

 const aiNodeAnim * findNodeAnim(const aiAnimation * p_animation, const string p_node_name)
 {
     // channel in animation contains aiNodeAnim (aiNodeAnim its transformation for bones)
     // numChannels == numBones
     for (uint i = 0; i < p_animation->mNumChannels; i++)
     {
         const aiNodeAnim* node_anim = p_animation->mChannels[i]; // ��������� ������� ������ node
         if (string(node_anim->mNodeName.data) == p_node_name)
         {
             return node_anim;// ���� ����� �������� �� ������� ����� (� ������� ����������� node) ������������ ���� node_anim
         }
     }

     return nullptr;
 }

 // start from RootNode
 void readNodeHierarchy(float p_animation_time, const aiNode* p_node, const aiMatrix4x4 parent_transform)
 {

     string node_name(p_node->mName.data);

     //������� node, �� ������� ������������ �������, ������������ �������� ���� ������(aiNodeAnim).
     const aiAnimation* animation = scene->mAnimations[0];
     aiMatrix4x4 node_transform = p_node->mTransformation; // AQUI AHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH

     const aiNodeAnim* node_anim = findNodeAnim(animation, node_name); // ����� ������� �� ����� ����

     if (node_anim)
     {

         //scaling
         //aiVector3D scaling_vector = node_anim->mScalingKeys[2].mValue;
         aiVector3D scaling_vector = calcInterpolatedScaling(p_animation_time, node_anim);
         aiMatrix4x4 scaling_matr;
         aiMatrix4x4::Scaling(scaling_vector, scaling_matr);

         //rotation
         //aiQuaternion rotate_quat = node_anim->mRotationKeys[2].mValue;
         aiQuaternion rotate_quat = calcInterpolatedRotation(p_animation_time, node_anim);
         aiMatrix4x4 rotate_matr = aiMatrix4x4(rotate_quat.GetMatrix());

         //translation
         //aiVector3D translate_vector = node_anim->mPositionKeys[2].mValue;
         aiVector3D translate_vector = calcInterpolatedPosition(p_animation_time, node_anim);
         aiMatrix4x4 translate_matr;
         aiMatrix4x4::Translation(translate_vector, translate_matr);

         //if (p_node->mName == scene->mRootNode->mName) {
         //  node_transform = translate_matr * (rotate_matr * aiMatrix4x4 (aiQuaternion(-90.0f, 0.0f, 0.0f).GetMatrix())) * scaling_matr;
         //}

         node_transform = translate_matr * rotate_matr * scaling_matr;
     }

     aiMatrix4x4 global_transform = parent_transform * node_transform;

     // ���� � node �� �������� ����������� bone, �� �� node ������ ��������� � ������ bone !!!
     if (m_bone_mapping.find(node_name) != m_bone_mapping.end()) // true if node_name exist in bone_mapping
     {
         uint bone_index = m_bone_mapping[node_name];
         m_bone_matrices[bone_index].final_world_transform = m_global_inverse_transform * global_transform * m_bone_matrices[bone_index].offset_matrix;
     }

     for (uint i = 0; i < p_node->mNumChildren; i++)
     {
         readNodeHierarchy(p_animation_time, p_node->mChildren[i], global_transform);
     }

 }

 void boneTransform(double time_in_sec, vector<aiMatrix4x4>& transforms)
 {
     aiMatrix4x4 identity_matrix; // = mat4(1.0f);
     double time_in_ticks = time_in_sec * ticks_per_second;
     float animation_time = fmod(time_in_ticks, scene->mAnimations[0]->mDuration); //������� �� ����� (������� �� ������)
     // animation_time - ���� ������� ������ � ���� ������ �� ������ �������� (�� ������� �������� ����� � �������� )

```

```
        readNodeHierarchy(animation_time, scene->mRootNode, identity_matrix);

        transforms.resize(m_num_bones);

        for (uint i = 0; i < m_num_bones; i++)
        {
            transforms[i] = m_bone_matrices[i].final_world_transform;
        }
    }

    glm::mat4 aiToGlm(aiMatrix4x4 ai_matr)
    {
        glm::mat4 result;
        result[0].x = ai_matr.a1; result[0].y = ai_matr.b1; result[0].z = ai_matr.c1; result[0].w = ai_matr.d1;
        result[1].x = ai_matr.a2; result[1].y = ai_matr.b2; result[1].z = ai_matr.c2; result[1].w = ai_matr.d2;
        result[2].x = ai_matr.a3; result[2].y = ai_matr.b3; result[2].z = ai_matr.c3; result[2].w = ai_matr.d3;
        result[3].x = ai_matr.a4; result[3].y = ai_matr.b4; result[3].z = ai_matr.c4; result[3].w = ai_matr.d4;

        //cout « " " « result[0].x « "        " « result[0].y « "       " « result[0].z « "       " « result[0].w « endl;
        //cout « " " « result[1].x « "        " « result[1].y « "       " « result[1].z « "       " « result[1].w « endl;
        //cout « " " « result[2].x « "        " « result[2].y « "       " « result[2].z « "       " « result[2].w « endl;
        //cout « " " « result[3].x « "        " « result[3].y « "       " « result[3].z « "       " « result[3].w « endl;
        //cout « endl;

        //cout « " " « ai_matr.a1 « "         " « ai_matr.b1 « "        " « ai_matr.c1 « "        " « ai_matr.d1 « endl;
        //cout « " " « ai_matr.a2 « "         " « ai_matr.b2 « "        " « ai_matr.c2 « "        " « ai_matr.d2 « endl;
        //cout « " " « ai_matr.a3 « "         " « ai_matr.b3 « "        " « ai_matr.c3 « "        " « ai_matr.d3 « endl;
        //cout « " " « ai_matr.a4 « "         " « ai_matr.b4 « "        " « ai_matr.c4 « "        " « ai_matr.d4 « endl;
        //cout « endl;

        return result;
    }


    aiQuaternion nlerp(aiQuaternion a, aiQuaternion b, float blend)
    {
        //cout « a.w + a.x + a.y + a.z « endl;
        a.Normalize();
        b.Normalize();

        aiQuaternion result;
        float dot_product = a.x * b.x + a.y * b.y + a.z * b.z + a.w * b.w;
        float one_minus_blend = 1.0f - blend;

        if (dot_product < 0.0f)
        {
            result.x = a.x * one_minus_blend + blend * -b.x;
            result.y = a.y * one_minus_blend + blend * -b.y;
            result.z = a.z * one_minus_blend + blend * -b.z;
            result.w = a.w * one_minus_blend + blend * -b.w;
        }
        else
        {
            result.x = a.x * one_minus_blend + blend * b.x;
            result.y = a.y * one_minus_blend + blend * b.y;
            result.z = a.z * one_minus_blend + blend * b.z;
            result.w = a.w * one_minus_blend + blend * b.w;
        }

        return result.Normalize();
    }
};
#endif
```

4.7 Shader.h

```
#ifndef SHADER_H
#define SHADER_H

#include <string>
#include <fstream>
```

```
#include <sstream>
#include <iostream>

#include <GL/glew.h>

class Shader
{
public:
    GLuint Program;
    GLuint uniformColor;
    // Constructor generates the shader on the fly
    Shader(const GLchar *vertexPath, const GLchar *fragmentPath)
    {
        // 1. Retrieve the vertex/fragment source code from filePath
        std::string vertexCode;
        std::string fragmentCode;
        std::ifstream vShaderFile;
        std::ifstream fShaderFile;
        // ensures ifstream objects can throw exceptions:
        vShaderFile.exceptions(std::ifstream::badbit);
        fShaderFile.exceptions(std::ifstream::badbit);
        try
        {
            // Open files
            vShaderFile.open(vertexPath);
            fShaderFile.open(fragmentPath);
            std::stringstream vShaderStream, fShaderStream;
            // Read file's buffer contents into streams
            vShaderStream « vShaderFile.rdbuf();
            fShaderStream « fShaderFile.rdbuf();
            // close file handlers
            vShaderFile.close();
            fShaderFile.close();
            // Convert stream into string
            vertexCode = vShaderStream.str();
            fragmentCode = fShaderStream.str();
        }
        catch (std::ifstream::failure e)
        {
            std::cout « "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" « std::endl;
        }
        const GLchar *vShaderCode = vertexCode.c_str();
        const GLchar *fShaderCode = fragmentCode.c_str();
        // 2. Compile shaders
        GLuint vertex, fragment;
        GLint success;
        GLchar infoLog[512];
        // Vertex Shader
        vertex = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(vertex, 1, &vShaderCode, NULL);
        glCompileShader(vertex);
        // Print compile errors if any
        glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
        if (!success)
        {
            glGetShaderInfoLog(vertex, 512, NULL, infoLog);
            std::cout « "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" « infoLog « std::endl;
        }
        // Fragment Shader
        fragment = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(fragment, 1, &fShaderCode, NULL);
        glCompileShader(fragment);
        // Print compile errors if any
        glGetShaderiv(fragment, GL_COMPILE_STATUS, &success);
        if (!success)
        {
            glGetShaderInfoLog(fragment, 512, NULL, infoLog);
            std::cout « "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" « infoLog « std::endl;
        }
        // Shader Program
        this->Program = glCreateProgram();
        glAttachShader(this->Program, vertex);
        glAttachShader(this->Program, fragment);
        glLinkProgram(this->Program);
        // Print linking errors if any
        glGetProgramiv(this->Program, GL_LINK_STATUS, &success);
        if (!success)
        {
            glGetProgramInfoLog(this->Program, 512, NULL, infoLog);
            std::cout « "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" « infoLog « std::endl;
        }
        //le damos la localidad de color
        uniformColor = glGetUniformLocation(this->Program, "color");
        // Delete the shaders as they're linked into our program now and no longer necessery
        glDeleteShader(vertex);
        glDeleteShader(fragment);

    }
    // Uses the current shader
    void Use()
    {
        glUseProgram(this->Program);
    }

     GLuint getColorLocation()
     {
         return uniformColor;
     }
};

#endif
```

4.8 stb_image.h

```
/* stb_image - v2.14 - public domain image loader - http://nothings.org/stb_image.h
no warranty implied; use at your own risk

Do this:
#define STB_IMAGE_IMPLEMENTATION
before you include this file in *one* C or C++ file to create the implementation.

// i.e. it should look like this:
#include ...
#include ...
#include ...
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

You can #define STBI_ASSERT(x) before the #include to avoid using assert.h.
And #define STBI_MALLOC, STBI_REALLOC, and STBI_FREE to avoid using malloc,realloc,free


QUICK NOTES:
Primarily of interest to game developers and other people who can
avoid problematic images and only need the trivial interface

JPEG baseline & progressive (12 bpc/arithmetic not supported, same as stock IJG lib)
PNG 1/2/4/8-bit-per-channel (16 bpc not supported)

TGA (not sure what subset, if a subset)
BMP non-1bpp, non-RLE
PSD (composited view only, no extra channels, 8/16 bit-per-channel)

GIF (*comp always reports as 4-channel)
HDR (radiance rgbE format)
PIC (Softimage PIC)
PNM (PPM and PGM binary only)

Animated GIF still needs a proper API, but here's one way to do it:
http://gist.github.com/urraka/685d9a6340b26b830d49

- decode from memory or through FILE (define STBI_NO_STDIO to remove code)
- decode from arbitrary I/O callbacks
- SIMD acceleration on x86/x64 (SSE2) and ARM (NEON)

Full documentation under "DOCUMENTATION" below.


Revision 2.00 release notes:

- Progressive JPEG is now supported.

- PPM and PGM binary formats are now supported, thanks to Ken Miller.

- x86 platforms now make use of SSE2 SIMD instructions for
JPEG decoding, and ARM platforms can use NEON SIMD if requested.
This work was done by Fabian "ryg" Giesen. SSE2 is used by
default, but NEON must be enabled explicitly; see docs.

With other JPEG optimizations included in this version, we see
2x speedup on a JPEG on an x86 machine, and a 1.5x speedup
on a JPEG on an ARM machine, relative to previous versions of this
library. The same results will not obtain for all JPGs and for all
x86/ARM machines. (Note that progressive JPEGs are significantly
slower to decode than regular JPEGs.) This doesn't mean that this
is the fastest JPEG decoder in the land; rather, it brings it
closer to parity with standard libraries. If you want the fastest
decode, look elsewhere. (See "Philosophy" section of docs below.)

See final bullet items below for more info on SIMD.
```

```

- Added STBI_MALLOC, STBI_REALLOC, and STBI_FREE macros for replacing
the memory allocator. Unlike other STBI libraries, these macros don't
support a context parameter, so if you need to pass a context in to
the allocator, you'll have to store it in a global or a thread-local
variable.

- Split existing STBI_NO_HDR flag into two flags, STBI_NO_HDR and
STBI_NO_LINEAR.
STBI_NO_HDR:     suppress implementation of .hdr reader format
STBI_NO_LINEAR:  suppress high-dynamic-range light-linear float API

- You can suppress implementation of any of the decoders to reduce
your code footprint by #defining one or more of the following
symbols before creating the implementation.

STBI_NO_JPEG
STBI_NO_PNG
STBI_NO_BMP
STBI_NO_PSD
STBI_NO_TGA
STBI_NO_GIF
STBI_NO_HDR
STBI_NO_PIC
STBI_NO_PNM   (.ppm and .pgm)

- You can request *only* certain decoders and suppress all other ones
(this will be more forward-compatible, as addition of new decoders
doesn't require you to disable them explicitly):

STBI_ONLY_JPEG
STBI_ONLY_PNG
STBI_ONLY_BMP
STBI_ONLY_PSD
STBI_ONLY_TGA
STBI_ONLY_GIF
STBI_ONLY_HDR
STBI_ONLY_PIC
STBI_ONLY_PNM   (.ppm and .pgm)

Note that you can define multiples of these, and you will get all
of them ("only x" and "only y" is interpreted to mean "only x&y").

- If you use STBI_NO_PNG (or _ONLY_ without PNG), and you still
want the zlib decoder to be available, #define STBI_SUPPORT_ZLIB

- Compilation of all SIMD code can be suppressed with
#define STBI_NO_SIMD
It should not be necessary to disable SIMD unless you have issues
compiling (e.g. using an x86 compiler which doesn't support SSE
intrinsics or that doesn't support the method used to detect
SSE2 support at run-time), and even those can be reported as
bugs so I can refine the built-in compile-time checking to be
smarter.

- The old STBI_SIMD system which allowed installing a user-defined
IDCT etc. has been removed. If you need this, don't upgrade. My
assumption is that almost nobody was doing this, and those who
were will find the built-in SIMD more satisfactory anyway.

- RGB values computed for JPEG images are slightly different from
previous versions of stb_image. (This is due to using less
integer precision in SIMD.) The C code has been adjusted so
that the same RGB values will be computed regardless of whether
SIMD support is available, so your app should always produce
consistent results. But these results are slightly different from
previous versions. (Specifically, about 3% of available YCbCr values
will compute different RGB results from pre-1.49 versions by +-1;
most of the deviating values are one smaller in the G channel.)

- If you must produce consistent results with previous versions of
stb_image, #define STBI_JPEG_OLD and you will get the same results
you used to; however, you will not get the SIMD speedups for
the YCbCr-to-RGB conversion step (although you should still see
significant JPEG speedup from the other changes).

Please note that STBI_JPEG_OLD is a temporary feature; it will be
removed in future versions of the library. It is only intended for
near-term back-compatibility use.


Latest revision history:
2.13  (2016-12-04) experimental 16-bit API, only for PNG so far; fixes
2.12  (2016-04-02) fix typo in 2.11 PSD fix that caused crashes
2.11  (2016-04-02) 16-bit PNGS; enable SSE2 in non-gcc x64
RGB-format JPEG; remove white matting in PSD;
allocate large structures on the stack;
```

```
correct channel count for PNG & BMP
2.10  (2016-01-22) avoid warning introduced in 2.09
2.09  (2016-01-16) 16-bit TGA; comments in PNM files; STBI_REALLOC_SIZED
2.08  (2015-09-13) fix to 2.07 cleanup, reading RGB PSD as RGBA
2.07  (2015-09-13) partial animated GIF support
limited 16-bit PSD support
minor bugs, code cleanup, and compiler warnings

See end of file for full revision history.


============================    Contributors    =========================

Image formats                          Extensions, features
Sean Barrett (jpeg, png, bmp)          Jetro Lauha (stbi_info)
Nicolas Schulz (hdr, psd)              Martin "SpartanJ" Golini (stbi_info)
Jonathan Dummer (tga)                  James "moose2000" Brown (iPhone PNG)
Jean-Marc Lienher (gif)                Ben "Disch" Wenger (io callbacks)
Tom Seddon (pic)                       Omar Cornut (1/2/4-bit PNG)
Thatcher Ulrich (psd)                  Nicolas Guillemot (vertical flip)
Ken Miller (pgm, ppm)                  Richard Mitton (16-bit PSD)
github:urraka (animated gif)           Junggon Kim (PNM comments)
Daniel Gibson (16-bit TGA)
socks-the-fox (16-bit TGA)
Optimizations & bugfixes
Fabian "ryg" Giesen
Arseny Kapoulkine

Bug & warning fixes
Marc LeBlanc            David Woo          Guillaume George   Martins Mozeiko
Christpher Lloyd        Martin Golini      Jerry Jansson      Joseph Thomson
Dave Moore              Roy Eltham         Hayaki Saito       Phil Jordan
Won Chun                Luke Graham        Johan Duparc       Nathan Reed
the Horde3D community   Thomas Ruf         Ronny Chevalier    Nick Verigakis
Janez Zemva             John Bartholomew   Michal Cichon      github:svdijk
Jonathan Blow           Ken Hamada         Tero Hanninen      Baldur Karlsson
Laurent Gomila          Cort Stratton      Sergio Gonzalez    github:romigrou
Aruelien Pocheville     Thibault Reuille   Cass Everitt       Matthew Gregan
Ryamond Barbiero        Paul Du Bois       Engin Manap        github:snagar
Michaelangel007@github  Oriol Ferrer Mesia Dale Weiler        github:Zelex
Philipp Wiesemann       Josh Tobin         github:rlyeh       github:grim210@github
Blazej Dariusz Roszkowski                  github:sammyhw


LICENSE

This software is dual-licensed to the public domain and under the following
license: you are granted a perpetual, irrevocable license to copy, modify,
publish, and distribute this file as you see fit.

*/

#ifndef STBI_INCLUDE_STB_IMAGE_H
#define STBI_INCLUDE_STB_IMAGE_H

// DOCUMENTATION
//
// Limitations:
//    - no 16-bit-per-channel PNG
//    - no 12-bit-per-channel JPEG
//    - no JPEGs with arithmetic coding
//    - no 1-bit BMP
//    - GIF always returns *comp=4
//
// Basic usage (see HDR discussion below for HDR usage):
//    int x,y,n;
//    unsigned char *data = stbi_load(filename, &x, &y, &n, 0);
//    // ... process data if not NULL ...
//    // ... x = width, y = height, n = # 8-bit components per pixel ...
//    // ... replace '0' with '1'..'4' to force that many components per pixel
//    // ... but 'n' will always be the number that it would have been if you said 0
//    stbi_image_free(data)
//
// Standard parameters:
//    int *x                 -- outputs image width in pixels
//    int *y                 -- outputs image height in pixels
//    int *channels_in_file  -- outputs # of image components in image file
//    int desired_channels   -- if non-zero, # of image components requested in result
//
// The return value from an image loader is an 'unsigned char *' which points
// to the pixel data, or NULL on an allocation failure or if the image is
// corrupt or invalid. The pixel data consists of *y scanlines of *x pixels,
// with each pixel consisting of N interleaved 8-bit components; the first
// pixel pointed to is top-left-most in the image. There is no padding between
// image scanlines or between pixels, regardless of format. The number of
// components N is 'req_comp' if req_comp is non-zero, or *comp otherwise.
// If req_comp is non-zero, *comp has the number of components that _would_
// have been output otherwise. E.g. if you set req_comp to 4, you will always
// get RGBA output, but you can check *comp to see if it's trivially opaque
// because e.g. there were only 3 channels in the source image.
//
// An output image with N components has the following components interleaved
// in this order in each pixel:
//
//     N=#comp     components
//       1           grey
//       2           grey, alpha
//       3           red, green, blue
//       4           red, green, blue, alpha
//
// If image loading fails for any reason, the return value will be NULL,
// and *x, *y, *comp will be unchanged. The function stbi_failure_reason()
// can be queried for an extremely brief, end-user unfriendly explanation
// of why the load failed. Define STBI_NO_FAILURE_STRINGS to avoid
// compiling these strings at all, and STBI_FAILURE_USERMSG to get slightly
// more user-friendly ones.
//
// Paletted PNG, BMP, GIF, and PIC images are automatically depalettized.
//
// ===========================================================================
//
// Philosophy
//
// stb libraries are designed with the following priorities:
//
//    1. easy to use
//    2. easy to maintain
//    3. good performance
//
// Sometimes I let "good performance" creep up in priority over "easy to maintain",
// and for best performance I may provide less-easy-to-use APIs that give higher
// performance, in addition to the easy to use ones. Nevertheless, it's important
// to keep in mind that from the standpoint of you, a client of this library,
// all you care about is #1 and #3, and stb libraries do not emphasize #3 above all.
//
// Some secondary priorities arise directly from the first two, some of which
// make more explicit reasons why performance can't be emphasized.
//
//    - Portable ("ease of use")
//    - Small footprint ("easy to maintain")
//    - No dependencies ("ease of use")
//
// ===========================================================================
//
// I/O callbacks
//
// I/O callbacks allow you to read from arbitrary sources, like packaged
// files or some other source. Data read from callbacks are processed
// through a small internal buffer (currently 128 bytes) to try to reduce
// overhead.
//
// The three functions you must define are "read" (reads some bytes of data),
// "skip" (skips some bytes of data), "eof" (reports if the stream is at the end).
//
// ===========================================================================
//
// SIMD support
//
// The JPEG decoder will try to automatically use SIMD kernels on x86 when
// supported by the compiler. For ARM Neon support, you must explicitly
// request it.
//
// (The old do-it-yourself SIMD API is no longer supported in the current
// code.)
//
// On x86, SSE2 will automatically be used when available based on a run-time
// test; if not, the generic C versions are used as a fall-back. On ARM targets,
// the typical path is to have separate builds for NEON and non-NEON devices
// (at least this is true for iOS and Android). Therefore, the NEON support is
// toggled by a build flag: define STBI_NEON to get NEON loops.
//
// The output of the JPEG decoder is slightly different from versions where
// SIMD support was introduced (that is, for versions before 1.49). The
// difference is only +-1 in the 8-bit RGB channels, and only on a small
// fraction of pixels. You can force the pre-1.49 behavior by defining
// STBI_JPEG_OLD, but this will disable some of the SIMD decoding path
// and hence cost some performance.
//
// If for some reason you do not want to use any of SIMD code, or if
// you have issues compiling it, you can disable it entirely by
// defining STBI_NO_SIMD.
//
// ===========================================================================
//
```

```
// HDR image support   (disable by defining STBI_NO_HDR)
//
// stb_image now supports loading HDR images in general, and currently
// the Radiance .HDR file format, although the support is provided
// generically. You can still load any file through the existing interface;
// if you attempt to load an HDR file, it will be automatically remapped to
// LDR, assuming gamma 2.2 and an arbitrary scale factor defaulting to 1;
// both of these constants can be reconfigured through this interface:
//
//     stbi_hdr_to_ldr_gamma(2.2f);
//     stbi_hdr_to_ldr_scale(1.0f);
//
// (note, do not use _inverse_ constants; stbi_image will invert them
// appropriately).
//
// Additionally, there is a new, parallel interface for loading files as
// (linear) floats to preserve the full dynamic range:
//
//    float *data = stbi_loadf(filename, &x, &y, &n, 0);
//
// If you load LDR images through this interface, those images will
// be promoted to floating point values, run through the inverse of
// constants corresponding to the above:
//
//     stbi_ldr_to_hdr_scale(1.0f);
//     stbi_ldr_to_hdr_gamma(2.2f);
//
// Finally, given a filename (or an open file or memory block--see header
// file for details) containing image data, you can query for the "most
// appropriate" interface to use (that is, whether the image is HDR or
// not), using:
//
//     stbi_is_hdr(char *filename);
//
// ===========================================================================
//
// iPhone PNG support:
//
// By default we convert iphone-formatted PNGs back to RGB, even though
// they are internally encoded differently. You can disable this conversion
// by by calling stbi_convert_iphone_png_to_rgb(0), in which case
// you will always just get the native iphone "format" through (which
// is BGR stored in RGB).
//
// Call stbi_set_unpremultiply_on_load(1) as well to force a divide per
// pixel to remove any premultiplied alpha *only* if the image file explicitly
// says there's premultiplied data (currently only happens in iPhone images,
// and only if iPhone convert-to-rgb processing is on).
//


#ifndef STBI_NO_STDIO
#include <stdio.h>
#endif // STBI_NO_STDIO

#define STBI_VERSION 1

enum
{
    STBI_default = 0, // only used for req_comp

    STBI_grey = 1,
    STBI_grey_alpha = 2,
    STBI_rgb = 3,
    STBI_rgb_alpha = 4
};

typedef unsigned char stbi_uc;
typedef unsigned short stbi_us;

#ifdef __cplusplus
extern "C" {
#endif

#ifdef STB_IMAGE_STATIC
#define STBIDEF static
#else
#define STBIDEF extern
#endif

    //
    // PRIMARY API - works on images of any type
    //

    //
    // load image by filename, open file, or memory buffer
    //

    typedef struct
    {
        int(*read)  (void *user, char *data, int size);   // fill 'data' with 'size' bytes.  return number of bytes actually read
        void(*skip)  (void *user, int n);                 // skip the next 'n' bytes, or 'unget' the last -n bytes if negative
        int(*eof)   (void *user);                       // returns nonzero if we are at end of file/data
    } stbi_io_callbacks;

    //
    // 8-bits-per-channel interface
    //

    STBIDEF stbi_uc *stbi_load(char              const *filename, int *x, int *y, int *channels_in_file, int desired_channels);
    STBIDEF stbi_uc *stbi_load_from_memory(stbi_uc          const *buffer, int len, int *x, int *y, int *channels_in_file, int desired_channels);
    STBIDEF stbi_uc *stbi_load_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *channels_in_file, int desired_channels);

#ifndef STBI_NO_STDIO
    STBIDEF stbi_uc *stbi_load_from_file(FILE *f, int *x, int *y, int *channels_in_file, int desired_channels);
    // for stbi_load_from_file, file pointer is left pointing immediately after image
#endif

    //
    // 16-bits-per-channel interface
    //

    STBIDEF stbi_us *stbi_load_16(char const *filename, int *x, int *y, int *channels_in_file, int desired_channels);
#ifndef STBI_NO_STDIO
    STBIDEF stbi_us *stbi_load_from_file_16(FILE *f, int *x, int *y, int *channels_in_file, int desired_channels);
#endif
    // @TODO the other variants

    //
    // float-per-channel interface
    //
#ifndef STBI_NO_LINEAR
    STBIDEF float *stbi_loadf(char const *filename, int *x, int *y, int *channels_in_file, int desired_channels);
    STBIDEF float *stbi_loadf_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *channels_in_file, int desired_channels);
    STBIDEF float *stbi_loadf_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *channels_in_file, int desired_channels);

#ifndef STBI_NO_STDIO
    STBIDEF float *stbi_loadf_from_file(FILE *f, int *x, int *y, int *channels_in_file, int desired_channels);
#endif
#endif

#ifndef STBI_NO_HDR
    STBIDEF void   stbi_hdr_to_ldr_gamma(float gamma);
    STBIDEF void   stbi_hdr_to_ldr_scale(float scale);
#endif // STBI_NO_HDR

#ifndef STBI_NO_LINEAR
    STBIDEF void   stbi_ldr_to_hdr_gamma(float gamma);
    STBIDEF void   stbi_ldr_to_hdr_scale(float scale);
#endif // STBI_NO_LINEAR

    // stbi_is_hdr is always defined, but always returns false if STBI_NO_HDR
    STBIDEF int    stbi_is_hdr_from_callbacks(stbi_io_callbacks const *clbk, void *user);
    STBIDEF int    stbi_is_hdr_from_memory(stbi_uc const *buffer, int len);
#ifndef STBI_NO_STDIO
    STBIDEF int      stbi_is_hdr(char const *filename);
    STBIDEF int      stbi_is_hdr_from_file(FILE *f);
#endif // STBI_NO_STDIO


    // get a VERY brief reason for failure
    // NOT THREADSAFE
    STBIDEF const char *stbi_failure_reason(void);

    // free the loaded image -- this is just free()
    STBIDEF void     stbi_image_free(void *retval_from_stbi_load);

    // get image dimensions & components without fully decoding
    STBIDEF int      stbi_info_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp);
    STBIDEF int      stbi_info_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y, int *comp);

```

```
#ifndef STBI_NO_STDIO
    STBIDEF int      stbi_info(char const *filename, int *x, int *y, int *comp);
    STBIDEF int      stbi_info_from_file(FILE *f, int *x, int *y, int *comp);

#endif



    // for image formats that explicitly notate that they have premultiplied alpha,
    // we just return the colors as stored in the file. set this flag to force
    // unpremultiplication. results are undefined if the unpremultiply overflow.
    STBIDEF void stbi_set_unpremultiply_on_load(int flag_true_if_should_unpremultiply);

    // indicate whether we should process iphone images back to canonical format,
    // or just pass them through "as-is"
    STBIDEF void stbi_convert_iphone_png_to_rgb(int flag_true_if_should_convert);

    // flip the image vertically, so the first pixel in the output array is the bottom left
    STBIDEF void stbi_set_flip_vertically_on_load(int flag_true_if_should_flip);

    // ZLIB client - used by PNG, available for other purposes

    STBIDEF char *stbi_zlib_decode_malloc_guesssize(const char *buffer, int len, int initial_size, int *outlen);
    STBIDEF char *stbi_zlib_decode_malloc_guesssize_headerflag(const char *buffer, int len, int initial_size, int *outlen, int parse_header);
    STBIDEF char *stbi_zlib_decode_malloc(const char *buffer, int len, int *outlen);
    STBIDEF int   stbi_zlib_decode_buffer(char *obuffer, int olen, const char *ibuffer, int ilen);

    STBIDEF char *stbi_zlib_decode_noheader_malloc(const char *buffer, int len, int *outlen);
    STBIDEF int   stbi_zlib_decode_noheader_buffer(char *obuffer, int olen, const char *ibuffer, int ilen);


#ifdef __cplusplus
}
#endif

//
//
#endif // STBI_INCLUDE_STB_IMAGE_H

#ifdef STB_IMAGE_IMPLEMENTATION

#if defined(STBI_ONLY_JPEG) || defined(STBI_ONLY_PNG) || defined(STBI_ONLY_BMP) \
  || defined(STBI_ONLY_TGA) || defined(STBI_ONLY_GIF) || defined(STBI_ONLY_PSD) \
  || defined(STBI_ONLY_HDR) || defined(STBI_ONLY_PIC) || defined(STBI_ONLY_PNM) \
  || defined(STBI_ONLY_ZLIB)
#ifndef STBI_ONLY_JPEG
#define STBI_NO_JPEG
#endif
#ifndef STBI_ONLY_PNG
#define STBI_NO_PNG
#endif
#ifndef STBI_ONLY_BMP
#define STBI_NO_BMP
#endif
#ifndef STBI_ONLY_PSD
#define STBI_NO_PSD
#endif
#ifndef STBI_ONLY_TGA
#define STBI_NO_TGA
#endif
#ifndef STBI_ONLY_GIF
#define STBI_NO_GIF
#endif
#ifndef STBI_ONLY_HDR
#define STBI_NO_HDR
#endif
#ifndef STBI_ONLY_PIC
#define STBI_NO_PIC
#endif
#ifndef STBI_ONLY_PNM
#define STBI_NO_PNM
#endif
#endif

#if defined(STBI_NO_PNG) && !defined(STBI_SUPPORT_ZLIB) && !defined(STBI_NO_ZLIB)
#define STBI_NO_ZLIB
#endif


#include <stdarg.h>
#include <stddef.h> // ptrdiff_t on osx
#include <stdlib.h>
#include <string.h>
#include <limits.h>
```

```

#if !defined(STBI_NO_LINEAR) || !defined(STBI_NO_HDR)
#include <math.h>  // ldexp
#endif

#ifndef STBI_NO_STDIO
#include <stdio.h>
#endif

#ifndef STBI_ASSERT
#include <assert.h>
#define STBI_ASSERT(x) assert(x)
#endif


#ifndef _MSC_VER
#ifdef __cplusplus
#define stbi_inline inline
#else
#define stbi_inline
#endif
#else
#define stbi_inline __forceinline
#endif


#ifdef _MSC_VER
typedef unsigned short stbi__uint16;
typedef   signed short stbi__int16;
typedef unsigned int   stbi__uint32;
typedef   signed int   stbi__int32;
#else
#include <stdint.h>
typedef uint16_t stbi__uint16;
typedef int16_t  stbi__int16;
typedef uint32_t stbi__uint32;
typedef int32_t  stbi__int32;
#endif

// should produce compiler error if size is wrong
typedef unsigned char validate_uint32[sizeof(stbi__uint32) == 4 ? 1 : -1];

#ifdef _MSC_VER
#define STBI_NOTUSED(v)  (void)(v)
#else
#define STBI_NOTUSED(v)  (void)sizeof(v)
#endif

#ifdef _MSC_VER
#define STBI_HAS_LROTL
#endif

#ifdef STBI_HAS_LROTL
#define stbi_lrot(x,y)  _lrotl(x,y)
#else
#define stbi_lrot(x,y)  (((x) « (y)) | ((x) » (32 - (y))))
#endif

#if defined(STBI_MALLOC) && defined(STBI_FREE) && (defined(STBI_REALLOC) || defined(STBI_REALLOC_SIZED))
// ok
#elif !defined(STBI_MALLOC) && !defined(STBI_FREE) && !defined(STBI_REALLOC) &&
   !defined(STBI_REALLOC_SIZED)
// ok
#else
#error "Must define all or none of STBI_MALLOC, STBI_FREE, and STBI_REALLOC (or STBI_REALLOC_SIZED)."
#endif

#ifndef STBI_MALLOC
#define STBI_MALLOC(sz)           malloc(sz)
#define STBI_REALLOC(p,newsz)     realloc(p,newsz)
#define STBI_FREE(p)              free(p)
#endif

#ifndef STBI_REALLOC_SIZED
#define STBI_REALLOC_SIZED(p,oldsz,newsz) STBI_REALLOC(p,newsz)
#endif

// x86/x64 detection
#if defined(__x86_64__) || defined(_M_X64)
#define STBI__X64_TARGET
#elif defined(__i386) || defined(_M_IX86)
#define STBI__X86_TARGET
#endif

#if defined(__GNUC__) && (defined(STBI__X86_TARGET) || defined(STBI__X64_TARGET)) && !defined(__SSE2__)
   && !defined(STBI_NO_SIMD)
// NOTE: not clear do we actually need this for the 64-bit path?
// gcc doesn’t support sse2 intrinsics unless you compile with -msse2,
// (but compiling with -msse2 allows the compiler to use SSE2 everywhere;
// this is just broken and gcc are jerks for not fixing it properly
// http://www.virtualdub.org/blog/pivot/entry.php?id=363 )
#define STBI_NO_SIMD
#endif

#if defined(__MINGW32__) && defined(STBI__X86_TARGET) && !defined(STBI_MINGW_ENABLE_SSE2) &&
   !defined(STBI_NO_SIMD)
// Note that __MINGW32__ doesn’t actually mean 32-bit, so we have to avoid STBI__X64_TARGET
//
// 32-bit MinGW wants ESP to be 16-byte aligned, but this is not in the
// Windows ABI and VC++ as well as Windows DLLs don’t maintain that invariant.
// As a result, enabling SSE2 on 32-bit MinGW is dangerous when not
// simultaneously enabling "-mstackrealign".
//
// See https://github.com/nothings/stb/issues/81 for more information.
//
// So default to no SSE2 on 32-bit MinGW. If you’ve read this far and added
// -mstackrealign to your build settings, feel free to #define STBI_MINGW_ENABLE_SSE2.
#define STBI_NO_SIMD
#endif

#if !defined(STBI_NO_SIMD) && (defined(STBI__X86_TARGET) || defined(STBI__X64_TARGET))
#define STBI_SSE2
#include <emmintrin.h>

#ifdef _MSC_VER

#if _MSC_VER >= 1400  // not VC6
#include <intrin.h> // __cpuid
static int stbi__cpuid3(void)
{
    int info[4];
    __cpuid(info, 1);
    return info[3];
}
#else
static int stbi__cpuid3(void)
{
    int res;
    __asm {
        mov  eax, 1
        cpuid
        mov  res, edx
    }
    return res;
}
#endif

#define STBI_SIMD_ALIGN(type, name) __declspec(align(16)) type name

static int stbi__sse2_available()
{
    int info3 = stbi__cpuid3();
    return ((info3 » 26) & 1) != 0;
}
#else // assume GCC-style if not VC++
#define STBI_SIMD_ALIGN(type, name) type name __attribute__((aligned(16)))

static int stbi__sse2_available()
{
#if defined(__GNUC__) && (__GNUC__ * 100 + __GNUC_MINOR__) >= 408 // GCC 4.8 or later
    // GCC 4.8+ has a nice way to do this
    return __builtin_cpu_supports("sse2");
#else
    // portable way to do this, preferably without using GCC inline ASM?
    // just bail for now.
    return 0;
#endif
}
#endif
#endif

// ARM NEON
#if defined(STBI_NO_SIMD) && defined(STBI_NEON)
#undef STBI_NEON
#endif

#ifdef STBI_NEON
#include <arm_neon.h>
// assume GCC or Clang on ARM targets
#define STBI_SIMD_ALIGN(type, name) type name __attribute__((aligned(16)))
#endif

#ifndef STBI_SIMD_ALIGN
#define STBI_SIMD_ALIGN(type, name) type name
```

```
#endif

//
//  stbi__context struct and start_xxx functions

// stbi__context structure is our basic context used by all images, so it
// contains all the IO context, plus some basic image information
typedef struct
{
    stbi__uint32 img_x, img_y;
    int img_n, img_out_n;

    stbi_io_callbacks io;
    void *io_user_data;

    int read_from_callbacks;
    int buflen;
    stbi_uc buffer_start[128];

    stbi_uc *img_buffer, *img_buffer_end;
    stbi_uc *img_buffer_original, *img_buffer_original_end;
} stbi__context;


static void stbi__refill_buffer(stbi__context *s);

// initialize a memory-decode context
static void stbi__start_mem(stbi__context *s, stbi_uc const *buffer, int len)
{
    s->io.read = NULL;
    s->read_from_callbacks = 0;
    s->img_buffer = s->img_buffer_original = (stbi_uc *)buffer;
    s->img_buffer_end = s->img_buffer_original_end = (stbi_uc *)buffer + len;
}

// initialize a callback-based context
static void stbi__start_callbacks(stbi__context *s, stbi_io_callbacks *c, void *user)
{
    s->io = *c;
    s->io_user_data = user;
    s->buflen = sizeof(s->buffer_start);
    s->read_from_callbacks = 1;
    s->img_buffer_original = s->buffer_start;
    stbi__refill_buffer(s);
    s->img_buffer_original_end = s->img_buffer_end;
}

#ifndef STBI_NO_STDIO

static int stbi__stdio_read(void *user, char *data, int size)
{
    return (int)fread(data, 1, size, (FILE*)user);
}

static void stbi__stdio_skip(void *user, int n)
{
    fseek((FILE*)user, n, SEEK_CUR);
}

static int stbi__stdio_eof(void *user)
{
    return feof((FILE*)user);
}

static stbi_io_callbacks stbi__stdio_callbacks =
{
    stbi__stdio_read,
    stbi__stdio_skip,
    stbi__stdio_eof,
};

static void stbi__start_file(stbi__context *s, FILE *f)
{
    stbi__start_callbacks(s, &stbi__stdio_callbacks, (void *)f);
}

//static void stop_file(stbi__context *s) { }

#endif // !STBI_NO_STDIO

static void stbi__rewind(stbi__context *s)
{
    // conceptually rewind SHOULD rewind to the beginning of the stream,
    // but we just rewind to the beginning of the initial buffer, because
    // we only use it after doing 'test', which only ever looks at at most 92 bytes
    s->img_buffer = s->img_buffer_original;
    s->img_buffer_end = s->img_buffer_original_end;
}

enum
{
    STBI_ORDER_RGB,
    STBI_ORDER_BGR
};

typedef struct
{
    int bits_per_channel;
    int num_channels;
    int channel_order;
} stbi__result_info;

#ifndef STBI_NO_JPEG
static int      stbi__jpeg_test(stbi__context *s);
static void    *stbi__jpeg_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri);
static int      stbi__jpeg_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_PNG
static int      stbi__png_test(stbi__context *s);
static void    *stbi__png_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri);
static int      stbi__png_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_BMP
static int      stbi__bmp_test(stbi__context *s);
static void    *stbi__bmp_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri);
static int      stbi__bmp_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_TGA
static int      stbi__tga_test(stbi__context *s);
static void    *stbi__tga_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri);
static int      stbi__tga_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_PSD
static int      stbi__psd_test(stbi__context *s);
static void    *stbi__psd_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri, int bpc);
static int      stbi__psd_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_HDR
static int      stbi__hdr_test(stbi__context *s);
static float   *stbi__hdr_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri);
static int      stbi__hdr_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_PIC
static int      stbi__pic_test(stbi__context *s);
static void    *stbi__pic_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri);
static int      stbi__pic_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_GIF
static int      stbi__gif_test(stbi__context *s);
static void    *stbi__gif_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri);
static int      stbi__gif_info(stbi__context *s, int *x, int *y, int *comp);
#endif

#ifndef STBI_NO_PNM
static int      stbi__pnm_test(stbi__context *s);
static void    *stbi__pnm_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri);
static int      stbi__pnm_info(stbi__context *s, int *x, int *y, int *comp);
#endif

// this is not threadsafe
static const char *stbi__g_failure_reason;

STBIDEF const char *stbi_failure_reason(void)
{
    return stbi__g_failure_reason;
}

static int stbi__err(const char *str)
```

```
{
    stbi__g_failure_reason = str;
    return 0;
}

static void *stbi__malloc(size_t size)
{
    return STBI_MALLOC(size);
}

// stb_image uses ints pervasively, including for offset calculations.
// therefore the largest decoded image size we can support with the
// current code, even on 64-bit targets, is INT_MAX. this is not a
// significant limitation for the intended use case.
//
// we do, however, need to make sure our size calculations don't
// overflow. hence a few helper functions for size calculations that
// multiply integers together, making sure that they're non-negative
// and no overflow occurs.

// return 1 if the sum is valid, 0 on overflow.
// negative terms are considered invalid.
static int stbi__addsizes_valid(int a, int b)
{
    if (b < 0) return 0;
    // now 0 <= b <= INT_MAX, hence also
    // 0 <= INT_MAX - b <= INTMAX.
    // And "a + b <= INT_MAX" (which might overflow) is the
    // same as a <= INT_MAX - b (no overflow)
    return a <= INT_MAX - b;
}

// returns 1 if the product is valid, 0 on overflow.
// negative factors are considered invalid.
static int stbi__mul2sizes_valid(int a, int b)
{
    if (a < 0 || b < 0) return 0;
    if (b == 0) return 1; // mul-by-0 is always safe
                          // portable way to check for no overflows in a*b
    return a <= INT_MAX / b;
}

// returns 1 if "a*b + add" has no negative terms/factors and doesn't overflow
static int stbi__mad2sizes_valid(int a, int b, int add)
{
    return stbi__mul2sizes_valid(a, b) && stbi__addsizes_valid(a*b, add);
}

// returns 1 if "a*b*c + add" has no negative terms/factors and doesn't overflow
static int stbi__mad3sizes_valid(int a, int b, int c, int add)
{
    return stbi__mul2sizes_valid(a, b) && stbi__mul2sizes_valid(a*b, c) &&
        stbi__addsizes_valid(a*b*c, add);
}

// returns 1 if "a*b*c*d + add" has no negative terms/factors and doesn't overflow
static int stbi__mad4sizes_valid(int a, int b, int c, int d, int add)
{
    return stbi__mul2sizes_valid(a, b) && stbi__mul2sizes_valid(a*b, c) &&
        stbi__mul2sizes_valid(a*b*c, d) && stbi__addsizes_valid(a*b*c*d, add);
}

// mallocs with size overflow checking
static void *stbi__malloc_mad2(int a, int b, int add)
{
    if (!stbi__mad2sizes_valid(a, b, add)) return NULL;
    return stbi__malloc(a*b + add);
}

static void *stbi__malloc_mad3(int a, int b, int c, int add)
{
    if (!stbi__mad3sizes_valid(a, b, c, add)) return NULL;
    return stbi__malloc(a*b*c + add);
}

static void *stbi__malloc_mad4(int a, int b, int c, int d, int add)
{
    if (!stbi__mad4sizes_valid(a, b, c, d, add)) return NULL;
    return stbi__malloc(a*b*c*d + add);
}

// stbi__err - error
// stbi__errpf - error returning pointer to float
// stbi__errpuc - error returning pointer to unsigned char

#ifdef STBI_NO_FAILURE_STRINGS
#define stbi__err(x,y)  0
```

```
#elif defined(STBI_FAILURE_USERMSG)
#define stbi__err(x,y)  stbi__err(y)
#else
#define stbi__err(x,y)  stbi__err(x)
#endif

#define stbi__errpf(x,y)   ((float *)(size_t) (stbi__err(x,y)?NULL:NULL))
#define stbi__errpuc(x,y)  ((unsigned char *)(size_t) (stbi__err(x,y)?NULL:NULL))

STBIDEF void stbi_image_free(void *retval_from_stbi_load)
{
    STBI_FREE(retval_from_stbi_load);
}

#ifndef STBI_NO_LINEAR
static float   *stbi__ldr_to_hdr(stbi_uc *data, int x, int y, int comp);
#endif

#ifndef STBI_NO_HDR
static stbi_uc *stbi__hdr_to_ldr(float   *data, int x, int y, int comp);
#endif

static int stbi__vertically_flip_on_load = 0;

STBIDEF void stbi_set_flip_vertically_on_load(int flag_true_if_should_flip)
{
    stbi__vertically_flip_on_load = flag_true_if_should_flip;
}

static void *stbi__load_main(stbi__context *s, int *x, int *y, int *comp, int req_comp,
  stbi__result_info *ri, int bpc)
{
    memset(ri, 0, sizeof(*ri)); // make sure it's initialized if we add new fields
    ri->bits_per_channel = 8; // default is 8 so most paths don't have to be changed
    ri->channel_order = STBI_ORDER_RGB; // all current input & output are this, but this is here so we
  can add BGR order
    ri->num_channels = 0;

#ifndef STBI_NO_JPEG
    if (stbi__jpeg_test(s)) return stbi__jpeg_load(s, x, y, comp, req_comp, ri);
#endif
#ifndef STBI_NO_PNG
    if (stbi__png_test(s))  return stbi__png_load(s, x, y, comp, req_comp, ri);
#endif
#ifndef STBI_NO_BMP
    if (stbi__bmp_test(s))  return stbi__bmp_load(s, x, y, comp, req_comp, ri);
#endif
#ifndef STBI_NO_GIF
    if (stbi__gif_test(s))  return stbi__gif_load(s, x, y, comp, req_comp, ri);
#endif
#ifndef STBI_NO_PSD
    if (stbi__psd_test(s))  return stbi__psd_load(s, x, y, comp, req_comp, ri, bpc);
#endif
#ifndef STBI_NO_PIC
    if (stbi__pic_test(s))  return stbi__pic_load(s, x, y, comp, req_comp, ri);
#endif
#ifndef STBI_NO_PNM
    if (stbi__pnm_test(s))  return stbi__pnm_load(s, x, y, comp, req_comp, ri);
#endif

#ifndef STBI_NO_HDR
    if (stbi__hdr_test(s)) {
        float *hdr = stbi__hdr_load(s, x, y, comp, req_comp, ri);
        return stbi__hdr_to_ldr(hdr, *x, *y, req_comp ? req_comp : *comp);
    }
#endif

#ifndef STBI_NO_TGA
    // test tga last because it's a crappy test!
    if (stbi__tga_test(s))
        return stbi__tga_load(s, x, y, comp, req_comp, ri);
#endif

    return stbi__errpuc("unknown image type", "Image not of any known type, or corrupt");
}

static stbi_uc *stbi__convert_16_to_8(stbi__uint16 *orig, int w, int h, int channels)
{
    int i;
    int img_len = w * h * channels;
    stbi_uc *reduced;

    reduced = (stbi_uc *)stbi__malloc(img_len);
    if (reduced == NULL) return stbi__errpuc("outofmem", "Out of memory");

    for (i = 0; i < img_len; ++i)
        reduced[i] = (stbi_uc)((orig[i] » 8) & 0xFF); // top half of each byte is sufficient approx of
  16->8 bit scaling

    STBI_FREE(orig);
    return reduced;
}

static stbi__uint16 *stbi__convert_8_to_16(stbi_uc *orig, int w, int h, int channels)
{
    int i;
    int img_len = w * h * channels;
    stbi__uint16 *enlarged;

    enlarged = (stbi__uint16 *)stbi__malloc(img_len * 2);
    if (enlarged == NULL) return (stbi__uint16 *)stbi__errpuc("outofmem", "Out of memory");

    for (i = 0; i < img_len; ++i)
        enlarged[i] = (stbi__uint16)((orig[i] « 8) + orig[i]); // replicate to high and low byte, maps
  0->0, 255->0xffff

    STBI_FREE(orig);
    return enlarged;
}

static unsigned char *stbi__load_and_postprocess_8bit(stbi__context *s, int *x, int *y, int *comp, int
  req_comp)
{
    stbi__result_info ri;
    void *result = stbi__load_main(s, x, y, comp, req_comp, &ri, 8);

    if (result == NULL)
        return NULL;

    if (ri.bits_per_channel != 8) {
        STBI_ASSERT(ri.bits_per_channel == 16);
        result = stbi__convert_16_to_8((stbi__uint16 *)result, *x, *y, req_comp == 0 ? *comp :
  req_comp);
        ri.bits_per_channel = 8;
    }

    // @TODO: move stbi__convert_format to here

    if (stbi__vertically_flip_on_load) {
        int w = *x, h = *y;
        int channels = req_comp ? req_comp : *comp;
        int row, col, z;
        stbi_uc *image = (stbi_uc *)result;

        // @OPTIMIZE: use a bigger temp buffer and memcpy multiple pixels at once
        for (row = 0; row < (h » 1); row++) {
            for (col = 0; col < w; col++) {
                for (z = 0; z < channels; z++) {
                    stbi_uc temp = image[(row * w + col) * channels + z];
                    image[(row * w + col) * channels + z] = image[((h - row - 1) * w + col) * channels
  + z];
                    image[((h - row - 1) * w + col) * channels + z] = temp;
                }
            }
        }
    }

    return (unsigned char *)result;
}

static stbi__uint16 *stbi__load_and_postprocess_16bit(stbi__context *s, int *x, int *y, int *comp, int
  req_comp)
{
    stbi__result_info ri;
    void *result = stbi__load_main(s, x, y, comp, req_comp, &ri, 16);

    if (result == NULL)
        return NULL;

    if (ri.bits_per_channel != 16) {
        STBI_ASSERT(ri.bits_per_channel == 8);
        result = stbi__convert_8_to_16((stbi_uc *)result, *x, *y, req_comp == 0 ? *comp : req_comp);
        ri.bits_per_channel = 16;
    }

    // @TODO: move stbi__convert_format16 to here
    // @TODO: special case RGB-to-Y (and RGBA-to-YA) for 8-bit-to-16-bit case to keep more precision

    if (stbi__vertically_flip_on_load) {
        int w = *x, h = *y;
        int channels = req_comp ? req_comp : *comp;
        int row, col, z;
        stbi__uint16 *image = (stbi__uint16 *)result;

        // @OPTIMIZE: use a bigger temp buffer and memcpy multiple pixels at once
        for (row = 0; row < (h » 1); row++) {
            for (col = 0; col < w; col++) {
                for (z = 0; z < channels; z++) {
                    stbi__uint16 temp = image[(row * w + col) * channels + z];
                    image[(row * w + col) * channels + z] = image[((h - row - 1) * w + col) * channels
    + z];
                    image[((h - row - 1) * w + col) * channels + z] = temp;
                }
            }
        }
    }

    return (stbi__uint16 *)result;
}

#ifndef STBI_NO_HDR
static void stbi__float_postprocess(float *result, int *x, int *y, int *comp, int req_comp)
{
    if (stbi__vertically_flip_on_load && result != NULL) {
        int w = *x, h = *y;
        int depth = req_comp ? req_comp : *comp;
        int row, col, z;
        float temp;

        // @OPTIMIZE: use a bigger temp buffer and memcpy multiple pixels at once
        for (row = 0; row < (h » 1); row++) {
            for (col = 0; col < w; col++) {
                for (z = 0; z < depth; z++) {
                    temp = result[(row * w + col) * depth + z];
                    result[(row * w + col) * depth + z] = result[((h - row - 1) * w + col) * depth +
    z];
                    result[((h - row - 1) * w + col) * depth + z] = temp;
                }
            }
        }
    }
}
#endif

#ifndef STBI_NO_STDIO

static FILE *stbi__fopen(char const *filename, char const *mode)
{
    FILE *f;
#if defined(_MSC_VER) && _MSC_VER >= 1400
    if (0 != fopen_s(&f, filename, mode))
        f = 0;
#else
    f = fopen(filename, mode);
#endif
    return f;
}


STBIDEF stbi_uc *stbi_load(char const *filename, int *x, int *y, int *comp, int req_comp)
{
    FILE *f = stbi__fopen(filename, "rb");
    unsigned char *result;
    if (!f) return stbi__errpuc("can't fopen", "Unable to open file");
    result = stbi_load_from_file(f, x, y, comp, req_comp);
    fclose(f);
    return result;
}

STBIDEF stbi_uc *stbi_load_from_file(FILE *f, int *x, int *y, int *comp, int req_comp)
{
    unsigned char *result;
    stbi__context s;
    stbi__start_file(&s, f);
    result = stbi__load_and_postprocess_8bit(&s, x, y, comp, req_comp);
    if (result) {
        // need to 'unget' all the characters in the IO buffer
        fseek(f, -(int)(s.img_buffer_end - s.img_buffer), SEEK_CUR);
    }
    return result;
}

STBIDEF stbi__uint16 *stbi_load_from_file_16(FILE *f, int *x, int *y, int *comp, int req_comp)
{
    stbi__uint16 *result;
    stbi__context s;
    stbi__start_file(&s, f);
    result = stbi__load_and_postprocess_16bit(&s, x, y, comp, req_comp);
    if (result) {
        // need to 'unget' all the characters in the IO buffer
        fseek(f, -(int)(s.img_buffer_end - s.img_buffer), SEEK_CUR);
    }
    return result;
}

STBIDEF stbi_us *stbi_load_16(char const *filename, int *x, int *y, int *comp, int req_comp)
{
    FILE *f = stbi__fopen(filename, "rb");
    stbi__uint16 *result;
    if (!f) return (stbi_us *)stbi__errpuc("can't fopen", "Unable to open file");
    result = stbi_load_from_file_16(f, x, y, comp, req_comp);
    fclose(f);
    return result;
}


#endif

STBIDEF stbi_uc *stbi_load_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp, int
   req_comp)
{
    stbi__context s;
    stbi__start_mem(&s, buffer, len);
    return stbi__load_and_postprocess_8bit(&s, x, y, comp, req_comp);
}

STBIDEF stbi_uc *stbi_load_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y,
   int *comp, int req_comp)
{
    stbi__context s;
    stbi__start_callbacks(&s, (stbi_io_callbacks *)clbk, user);
    return stbi__load_and_postprocess_8bit(&s, x, y, comp, req_comp);
}

#ifndef STBI_NO_LINEAR
static float *stbi__loadf_main(stbi__context *s, int *x, int *y, int *comp, int req_comp)
{
    unsigned char *data;
#ifndef STBI_NO_HDR
    if (stbi__hdr_test(s)) {
        stbi__result_info ri;
        float *hdr_data = stbi__hdr_load(s, x, y, comp, req_comp, &ri);
        if (hdr_data)
            stbi__float_postprocess(hdr_data, x, y, comp, req_comp);
        return hdr_data;
    }
#endif
    data = stbi__load_and_postprocess_8bit(s, x, y, comp, req_comp);
    if (data)
        return stbi__ldr_to_hdr(data, *x, *y, req_comp ? req_comp : *comp);
    return stbi__errpf("unknown image type", "Image not of any known type, or corrupt");
}

STBIDEF float *stbi_loadf_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp, int
   req_comp)
{
    stbi__context s;
    stbi__start_mem(&s, buffer, len);
    return stbi__loadf_main(&s, x, y, comp, req_comp);
}

STBIDEF float *stbi_loadf_from_callbacks(stbi_io_callbacks const *clbk, void *user, int *x, int *y, int
   *comp, int req_comp)
{
    stbi__context s;
    stbi__start_callbacks(&s, (stbi_io_callbacks *)clbk, user);
    return stbi__loadf_main(&s, x, y, comp, req_comp);
}

#ifndef STBI_NO_STDIO
STBIDEF float *stbi_loadf(char const *filename, int *x, int *y, int *comp, int req_comp)
{
    float *result;
    FILE *f = stbi__fopen(filename, "rb");
    if (!f) return stbi__errpf("can't fopen", "Unable to open file");
    result = stbi_loadf_from_file(f, x, y, comp, req_comp);
    fclose(f);
    return result;
}

STBIDEF float *stbi_loadf_from_file(FILE *f, int *x, int *y, int *comp, int req_comp)
{
    stbi__context s;
    stbi__start_file(&s, f);
    return stbi__loadf_main(&s, x, y, comp, req_comp);
}
#endif // !STBI_NO_STDIO

```

```
#endif // !STBI_NO_LINEAR

// these is-hdr-or-not is defined independent of whether STBI_NO_LINEAR is
// defined, for API simplicity; if STBI_NO_LINEAR is defined, it always
// reports false!

STBIDEF int stbi_is_hdr_from_memory(stbi_uc const *buffer, int len)
{
#ifndef STBI_NO_HDR
    stbi__context s;
    stbi__start_mem(&s, buffer, len);
    return stbi__hdr_test(&s);
#else
    STBI_NOTUSED(buffer);
    STBI_NOTUSED(len);
    return 0;
#endif
}

#ifndef STBI_NO_STDIO
STBIDEF int      stbi_is_hdr(char const *filename)
{
    FILE *f = stbi__fopen(filename, "rb");
    int result = 0;
    if (f) {
        result = stbi_is_hdr_from_file(f);
        fclose(f);
    }
    return result;
}

STBIDEF int      stbi_is_hdr_from_file(FILE *f)
{
#ifndef STBI_NO_HDR
    stbi__context s;
    stbi__start_file(&s, f);
    return stbi__hdr_test(&s);
#else
    STBI_NOTUSED(f);
    return 0;
#endif
}
#endif // !STBI_NO_STDIO

STBIDEF int      stbi_is_hdr_from_callbacks(stbi_io_callbacks const *clbk, void *user)
{
#ifndef STBI_NO_HDR
    stbi__context s;
    stbi__start_callbacks(&s, (stbi_io_callbacks *)clbk, user);
    return stbi__hdr_test(&s);
#else
    STBI_NOTUSED(clbk);
    STBI_NOTUSED(user);
    return 0;
#endif
}

#ifndef STBI_NO_LINEAR
static float stbi__l2h_gamma = 2.2f, stbi__l2h_scale = 1.0f;

STBIDEF void   stbi_ldr_to_hdr_gamma(float gamma) { stbi__l2h_gamma = gamma; }
STBIDEF void   stbi_ldr_to_hdr_scale(float scale) { stbi__l2h_scale = scale; }
#endif

static float stbi__h2l_gamma_i = 1.0f / 2.2f, stbi__h2l_scale_i = 1.0f;

STBIDEF void   stbi_hdr_to_ldr_gamma(float gamma) { stbi__h2l_gamma_i = 1 / gamma; }
STBIDEF void   stbi_hdr_to_ldr_scale(float scale) { stbi__h2l_scale_i = 1 / scale; }


//
// Common code used by all image loaders
//

enum
{
    STBI__SCAN_load = 0,
    STBI__SCAN_type,
    STBI__SCAN_header
};

static void stbi__refill_buffer(stbi__context *s)
{
    int n = (s->io.read)(s->io_user_data, (char*)s->buffer_start, s->buflen);
    if (n == 0) {
        // at end of file, treat same as if from memory, but need to handle case
        // where s->img_buffer isn't pointing to safe memory, e.g. 0-byte file
        s->read_from_callbacks = 0;
        s->img_buffer = s->buffer_start;
        s->img_buffer_end = s->buffer_start + 1;
        *s->img_buffer = 0;
    }
    else {
        s->img_buffer = s->buffer_start;
        s->img_buffer_end = s->buffer_start + n;
    }
}

stbi_inline static stbi_uc stbi__get8(stbi__context *s)
{
    if (s->img_buffer < s->img_buffer_end)
        return *s->img_buffer++;
    if (s->read_from_callbacks) {
        stbi__refill_buffer(s);
        return *s->img_buffer++;
    }
    return 0;
}

stbi_inline static int stbi__at_eof(stbi__context *s)
{
    if (s->io.read) {
        if (!(s->io.eof)(s->io_user_data)) return 0;
        // if feof() is true, check if buffer = end
        // special case: we've only got the special 0 character at the end
        if (s->read_from_callbacks == 0) return 1;
    }

    return s->img_buffer >= s->img_buffer_end;
}

static void stbi__skip(stbi__context *s, int n)
{
    if (n < 0) {
        s->img_buffer = s->img_buffer_end;
        return;
    }
    if (s->io.read) {
        int blen = (int)(s->img_buffer_end - s->img_buffer);
        if (blen < n) {
            s->img_buffer = s->img_buffer_end;
            (s->io.skip)(s->io_user_data, n - blen);
            return;
        }
    }
    s->img_buffer += n;
}

static int stbi__getn(stbi__context *s, stbi_uc *buffer, int n)
{
    if (s->io.read) {
        int blen = (int)(s->img_buffer_end - s->img_buffer);
        if (blen < n) {
            int res, count;

            memcpy(buffer, s->img_buffer, blen);

            count = (s->io.read)(s->io_user_data, (char*)buffer + blen, n - blen);
            res = (count == (n - blen));
            s->img_buffer = s->img_buffer_end;
            return res;
        }
    }

    if (s->img_buffer + n <= s->img_buffer_end) {
        memcpy(buffer, s->img_buffer, n);
        s->img_buffer += n;
        return 1;
    }
    else
        return 0;
}

static int stbi__get16be(stbi__context *s)
{
    int z = stbi__get8(s);
    return (z « 8) + stbi__get8(s);
}

static stbi__uint32 stbi__get32be(stbi__context *s)
{
    stbi__uint32 z = stbi__get16be(s);
    return (z « 16) + stbi__get16be(s);
}
```

```

#if defined(STBI_NO_BMP) && defined(STBI_NO_TGA) && defined(STBI_NO_GIF)
// nothing
#else
static int stbi__get16le(stbi__context *s)
{
    int z = stbi__get8(s);
    return z + (stbi__get8(s) « 8);
}
#endif

#ifndef STBI_NO_BMP
static stbi__uint32 stbi__get32le(stbi__context *s)
{
    stbi__uint32 z = stbi__get16le(s);
    return z + (stbi__get16le(s) « 16);
}
#endif

#define STBI__BYTECAST(x)  ((stbi_uc) ((x) & 255))  // truncate int to byte without warnings


//
//  generic converter from built-in img_n to req_comp
//    individual types do this automatically as much as possible (e.g. jpeg
//    does all cases internally since it needs to colorspace convert anyway,
//    and it never has alpha, so very few cases ). png can automatically
//    interleave an alpha=255 channel, but falls back to this for other cases
//
//  assume data buffer is malloced, so malloc a new one and free that one
//  only failure mode is malloc failing

static stbi_uc stbi__compute_y(int r, int g, int b)
{
    return (stbi_uc)(((r * 77) + (g * 150) + (29 * b)) » 8);
}

static unsigned char *stbi__convert_format(unsigned char *data, int img_n, int req_comp, unsigned int
   x, unsigned int y)
{
    int i, j;
    unsigned char *good;

    if (req_comp == img_n) return data;
    STBI_ASSERT(req_comp >= 1 && req_comp <= 4);

    good = (unsigned char *)stbi__malloc_mad3(req_comp, x, y, 0);
    if (good == NULL) {
        STBI_FREE(data);
        return stbi__errpuc("outofmem", "Out of memory");
    }

    for (j = 0; j < (int)y; ++j) {
        unsigned char *src = data + j * x * img_n;
        unsigned char *dest = good + j * x * req_comp;

#define STBI__COMBO(a,b)  ((a)*8+(b))
#define STBI__CASE(a,b)   case STBI__COMBO(a,b): for(i=x-1; i >= 0; --i, src += a, dest += b)
        // convert source image with img_n components to one with req_comp components;
        // avoid switch per pixel, so use switch per scanline and massive macros
        switch (STBI__COMBO(img_n, req_comp)) {
            STBI__CASE(1, 2) { dest[0] = src[0], dest[1] = 255; } break;
            STBI__CASE(1, 3) { dest[0] = dest[1] = dest[2] = src[0]; } break;
            STBI__CASE(1, 4) { dest[0] = dest[1] = dest[2] = src[0], dest[3] = 255; } break;
            STBI__CASE(2, 1) { dest[0] = src[0]; } break;
            STBI__CASE(2, 3) { dest[0] = dest[1] = dest[2] = src[0]; } break;
            STBI__CASE(2, 4) { dest[0] = dest[1] = dest[2] = src[0], dest[3] = src[1]; } break;
            STBI__CASE(3, 4) { dest[0] = src[0], dest[1] = src[1], dest[2] = src[2], dest[3] = 255; }
   break;
            STBI__CASE(3, 1) { dest[0] = stbi__compute_y(src[0], src[1], src[2]); } break;
            STBI__CASE(3, 2) { dest[0] = stbi__compute_y(src[0], src[1], src[2]), dest[1] = 255; }
   break;
            STBI__CASE(4, 1) { dest[0] = stbi__compute_y(src[0], src[1], src[2]); } break;
            STBI__CASE(4, 2) { dest[0] = stbi__compute_y(src[0], src[1], src[2]), dest[1] = src[3]; }
   break;
            STBI__CASE(4, 3) { dest[0] = src[0], dest[1] = src[1], dest[2] = src[2]; } break;
        default: STBI_ASSERT(0);
        }
#undef STBI__CASE
    }

    STBI_FREE(data);
    return good;
}

static stbi__uint16 stbi__compute_y_16(int r, int g, int b)
{
```

```c
    return (stbi__uint16)(((r * 77) + (g * 150) + (29 * b)) >> 8);
}

static stbi__uint16 *stbi__convert_format16(stbi__uint16 *data, int img_n, int req_comp, unsigned int x, unsigned int y)
{
    int i, j;
    stbi__uint16 *good;

    if (req_comp == img_n) return data;
    STBI_ASSERT(req_comp >= 1 && req_comp <= 4);

    good = (stbi__uint16 *)stbi__malloc(req_comp * x * y * 2);
    if (good == NULL) {
        STBI_FREE(data);
        return (stbi__uint16 *)stbi__errpuc("outofmem", "Out of memory");
    }

    for (j = 0; j < (int)y; ++j) {
        stbi__uint16 *src = data + j * x * img_n;
        stbi__uint16 *dest = good + j * x * req_comp;

#define STBI__COMBO(a,b)  ((a)*8+(b))
#define STBI__CASE(a,b)   case STBI__COMBO(a,b): for(i=x-1; i >= 0; --i, src += a, dest += b)
        // convert source image with img_n components to one with req_comp components;
        // avoid switch per pixel, so use switch per scanline and massive macros
        switch (STBI__COMBO(img_n, req_comp)) {
            STBI__CASE(1, 2) { dest[0] = src[0], dest[1] = 0xffff; } break;
            STBI__CASE(1, 3) { dest[0] = dest[1] = dest[2] = src[0]; } break;
            STBI__CASE(1, 4) { dest[0] = dest[1] = dest[2] = src[0], dest[3] = 0xffff; } break;
            STBI__CASE(2, 1) { dest[0] = src[0]; } break;
            STBI__CASE(2, 3) { dest[0] = dest[1] = dest[2] = src[0]; } break;
            STBI__CASE(2, 4) { dest[0] = dest[1] = dest[2] = src[0], dest[3] = src[1]; } break;
            STBI__CASE(3, 4) { dest[0] = src[0], dest[1] = src[1], dest[2] = src[2], dest[3] = 0xffff; } break;
            STBI__CASE(3, 1) { dest[0] = stbi__compute_y_16(src[0], src[1], src[2]); } break;
            STBI__CASE(3, 2) { dest[0] = stbi__compute_y_16(src[0], src[1], src[2]), dest[1] = 0xffff; } break;
            STBI__CASE(4, 1) { dest[0] = stbi__compute_y_16(src[0], src[1], src[2]); } break;
            STBI__CASE(4, 2) { dest[0] = stbi__compute_y_16(src[0], src[1], src[2]), dest[1] = src[3]; } break;
            STBI__CASE(4, 3) { dest[0] = src[0], dest[1] = src[1], dest[2] = src[2]; } break;
        default: STBI_ASSERT(0);
        }
#undef STBI__CASE
    }

    STBI_FREE(data);
    return good;
}

#ifndef STBI_NO_LINEAR
static float   *stbi__ldr_to_hdr(stbi_uc *data, int x, int y, int comp)
{
    int i, k, n;
    float *output;
    if (!data) return NULL;
    output = (float *)stbi__malloc_mad4(x, y, comp, sizeof(float), 0);
    if (output == NULL) { STBI_FREE(data); return stbi__errpf("outofmem", "Out of memory"); }
    // compute number of non-alpha components
    if (comp & 1) n = comp; else n = comp - 1;
    for (i = 0; i < x*y; ++i) {
        for (k = 0; k < n; ++k) {
            output[i*comp + k] = (float)(pow(data[i*comp + k] / 255.0f, stbi__l2h_gamma) * stbi__l2h_scale);
        }
        if (k < comp) output[i*comp + k] = data[i*comp + k] / 255.0f;
    }
    STBI_FREE(data);
    return output;
}
#endif

#ifndef STBI_NO_HDR
#define stbi__float2int(x)   ((int) (x))
static stbi_uc *stbi__hdr_to_ldr(float   *data, int x, int y, int comp)
{
    int i, k, n;
    stbi_uc *output;
    if (!data) return NULL;
    output = (stbi_uc *)stbi__malloc_mad3(x, y, comp, 0);
    if (output == NULL) { STBI_FREE(data); return stbi__errpuc("outofmem", "Out of memory"); }
    // compute number of non-alpha components
    if (comp & 1) n = comp; else n = comp - 1;
    for (i = 0; i < x*y; ++i) {
        for (k = 0; k < n; ++k) {
            float z = (float)pow(data[i*comp + k] * stbi__h2l_scale_i, stbi__h2l_gamma_i) * 255 + 0.5f;
            if (z < 0) z = 0;
            if (z > 255) z = 255;
            output[i*comp + k] = (stbi_uc)stbi__float2int(z);
        }
        if (k < comp) {
            float z = data[i*comp + k] * 255 + 0.5f;
            if (z < 0) z = 0;
            if (z > 255) z = 255;
            output[i*comp + k] = (stbi_uc)stbi__float2int(z);
        }
    }
    STBI_FREE(data);
    return output;
}
#endif

//
//  "baseline" JPEG/JFIF decoder
//
//    simple implementation
//      - doesn't support delayed output of y-dimension
//      - simple interface (only one output format: 8-bit interleaved RGB)
//      - doesn't try to recover corrupt jpegs
//      - doesn't allow partial loading, loading multiple at once
//      - still fast on x86 (copying globals into locals doesn't help x86)
//      - allocates lots of intermediate memory (full size of all components)
//        - non-interleaved case requires this anyway
//        - allows good upsampling (see next)
//    high-quality
//      - upsampled channels are bilinearly interpolated, even across blocks
//      - quality integer IDCT derived from IJG's 'slow'
//    performance
//      - fast huffman; reasonable integer IDCT
//      - some SIMD kernels for common paths on targets with SSE2/NEON
//      - uses a lot of intermediate memory, could cache poorly

#ifndef STBI_NO_JPEG

// huffman decoding acceleration
#define FAST_BITS   9  // larger handles more cases; smaller stomps less cache

typedef struct
{
    stbi_uc  fast[1 « FAST_BITS];
    // weirdly, repacking this into AoS is a 10% speed loss, instead of a win
    stbi__uint16 code[256];
    stbi_uc  values[256];
    stbi_uc  size[257];
    unsigned int maxcode[18];
    int    delta[17];   // old 'firstsymbol' - old 'firstcode'
} stbi__huffman;

typedef struct
{
    stbi__context *s;
    stbi__huffman huff_dc[4];
    stbi__huffman huff_ac[4];
    stbi_uc dequant[4][64];
    stbi__int16 fast_ac[4][1 « FAST_BITS];

    // sizes for components, interleaved MCUs
    int img_h_max, img_v_max;
    int img_mcu_x, img_mcu_y;
    int img_mcu_w, img_mcu_h;

    // definition of jpeg image component
    struct
    {
        int id;
        int h, v;
        int tq;
        int hd, ha;
        int dc_pred;

        int x, y, w2, h2;
        stbi_uc *data;
        void *raw_data, *raw_coeff;
        stbi_uc *linebuf;
        short   *coeff;   // progressive only
        int      coeff_w, coeff_h; // number of 8x8 coefficient blocks
    } img_comp[4];

    stbi__uint32   code_buffer; // jpeg entropy-coded buffer
    int            code_bits;   // number of valid bits
    unsigned char  marker;      // marker seen while filling entropy buffer
    int            nomore;      // flag if we saw a marker so must stop

```

```
   int              progressive;
   int              spec_start;
   int              spec_end;
   int              succ_high;
   int              succ_low;
   int              eob_run;
   int              rgb;

   int scan_n, order[4];
   int restart_interval, todo;

   // kernels
   void(*idct_block_kernel)(stbi_uc *out, int out_stride, short data[64]);
   void(*YCbCr_to_RGB_kernel)(stbi_uc *out, const stbi_uc *y, const stbi_uc *pcb, const stbi_uc *pcr,
 int count, int step);
   stbi_uc *(*resample_row_hv_2_kernel)(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int
 hs);
} stbi__jpeg;

static int stbi__build_huffman(stbi__huffman *h, int *count)
{
   int i, j, k = 0, code;
   // build size list for each symbol (from JPEG spec)
   for (i = 0; i < 16; ++i)
      for (j = 0; j < count[i]; ++j)
         h->size[k++] = (stbi_uc)(i + 1);
   h->size[k] = 0;

   // compute actual symbols (from jpeg spec)
   code = 0;
   k = 0;
   for (j = 1; j <= 16; ++j) {
      // compute delta to add to code to compute symbol id
      h->delta[j] = k - code;
      if (h->size[k] == j) {
         while (h->size[k] == j)
            h->code[k++] = (stbi__uint16)(code++);
         if (code - 1 >= (1 « j)) return stbi__err("bad code lengths", "Corrupt JPEG");
      }
      // compute largest code + 1 for this size, preshifted as needed later
      h->maxcode[j] = code « (16 - j);
      code «= 1;
   }
   h->maxcode[j] = 0xffffffff;

   // build non-spec acceleration table; 255 is flag for not-accelerated
   memset(h->fast, 255, 1 « FAST_BITS);
   for (i = 0; i < k; ++i) {
      int s = h->size[i];
      if (s <= FAST_BITS) {
         int c = h->code[i] « (FAST_BITS - s);
         int m = 1 « (FAST_BITS - s);
         for (j = 0; j < m; ++j) {
            h->fast[c + j] = (stbi_uc)i;
         }
      }
   }
   return 1;
}

// build a table that decodes both magnitude and value of small ACs in
// one go.
static void stbi__build_fast_ac(stbi__int16 *fast_ac, stbi__huffman *h)
{
   int i;
   for (i = 0; i < (1 « FAST_BITS); ++i) {
      stbi_uc fast = h->fast[i];
      fast_ac[i] = 0;
      if (fast < 255) {
         int rs = h->values[fast];
         int run = (rs » 4) & 15;
         int magbits = rs & 15;
         int len = h->size[fast];

         if (magbits && len + magbits <= FAST_BITS) {
            // magnitude code followed by receive_extend code
            int k = ((i « len) & ((1 « FAST_BITS) - 1)) » (FAST_BITS - magbits);
            int m = 1 « (magbits - 1);
            if (k < m) k += (-1 « magbits) + 1;
            // if the result is small enough, we can fit it in fast_ac table
            if (k >= -128 && k <= 127)
               fast_ac[i] = (stbi__int16)((k « 8) + (run « 4) + (len + magbits));
         }
      }
   }
}

```

```
static void stbi__grow_buffer_unsafe(stbi__jpeg *j)
{
    do {
        int b = j->nomore ? 0 : stbi__get8(j->s);
        if (b == 0xff) {
            int c = stbi__get8(j->s);
            if (c != 0) {
                j->marker = (unsigned char)c;
                j->nomore = 1;
                return;
            }
        }
        j->code_buffer |= b « (24 - j->code_bits);
        j->code_bits += 8;
    } while (j->code_bits <= 24);
}

// (1 « n) - 1
static stbi__uint32 stbi__bmask[17] = {
  0,1,3,7,15,31,63,127,255,511,1023,2047,4095,8191,16383,32767,65535 };

// decode a jpeg huffman value from the bitstream
stbi_inline static int stbi__jpeg_huff_decode(stbi__jpeg *j, stbi__huffman *h)
{
    unsigned int temp;
    int c, k;

    if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);

    // look at the top FAST_BITS and determine what symbol ID it is,
    // if the code is <= FAST_BITS
    c = (j->code_buffer » (32 - FAST_BITS)) & ((1 « FAST_BITS) - 1);
    k = h->fast[c];
    if (k < 255) {
        int s = h->size[k];
        if (s > j->code_bits)
            return -1;
        j->code_buffer «= s;
        j->code_bits -= s;
        return h->values[k];
    }

    // naive test is to shift the code_buffer down so k bits are
    // valid, then test against maxcode. To speed this up, we've
    // preshifted maxcode left so that it has (16-k) 0s at the
    // end; in other words, regardless of the number of bits, it
    // wants to be compared against something shifted to have 16;
    // that way we don't need to shift inside the loop.
    temp = j->code_buffer » 16;
    for (k = FAST_BITS + 1; ; ++k)
        if (temp < h->maxcode[k])
            break;
    if (k == 17) {
        // error! code not found
        j->code_bits -= 16;
        return -1;
    }

    if (k > j->code_bits)
        return -1;

    // convert the huffman code to the symbol id
    c = ((j->code_buffer » (32 - k)) & stbi__bmask[k]) + h->delta[k];
    STBI_ASSERT((((j->code_buffer) » (32 - h->size[c])) & stbi__bmask[h->size[c]]) == h->code[c]);

    // convert the id to a symbol
    j->code_bits -= k;
    j->code_buffer «= k;
    return h->values[c];
}

// bias[n] = (-1«n) + 1
static int const stbi__jbias[16] = {
  0,-1,-3,-7,-15,-31,-63,-127,-255,-511,-1023,-2047,-4095,-8191,-16383,-32767 };

// combined JPEG 'receive' and JPEG 'extend', since baseline
// always extends everything it receives.
stbi_inline static int stbi__extend_receive(stbi__jpeg *j, int n)
{
    unsigned int k;
    int sgn;
    if (j->code_bits < n) stbi__grow_buffer_unsafe(j);

    sgn = (stbi__int32)j->code_buffer » 31; // sign bit is always in MSB
    k = stbi_lrot(j->code_buffer, n);
    STBI_ASSERT(n >= 0 && n < (int)(sizeof(stbi__bmask) / sizeof(*stbi__bmask)));
    j->code_buffer = k & ~stbi__bmask[n];
    k &= stbi__bmask[n];
    j->code_bits -= n;
    return k + (stbi__jbias[n] & ~sgn);
}

// get some unsigned bits
stbi_inline static int stbi__jpeg_get_bits(stbi__jpeg *j, int n)
{
    unsigned int k;
    if (j->code_bits < n) stbi__grow_buffer_unsafe(j);
    k = stbi_lrot(j->code_buffer, n);
    j->code_buffer = k & ~stbi__bmask[n];
    k &= stbi__bmask[n];
    j->code_bits -= n;
    return k;
}

stbi_inline static int stbi__jpeg_get_bit(stbi__jpeg *j)
{
    unsigned int k;
    if (j->code_bits < 1) stbi__grow_buffer_unsafe(j);
    k = j->code_buffer;
    j->code_buffer «= 1;
    --j->code_bits;
    return k & 0x80000000;
}

// given a value that's at position X in the zigzag stream,
// where does it appear in the 8x8 matrix coded as row-major?
static stbi_uc stbi__jpeg_dezigzag[64 + 15] =
{
    0,  1,  8, 16,  9,  2,  3, 10,
    17, 24, 32, 25, 18, 11,  4,  5,
    12, 19, 26, 33, 40, 48, 41, 34,
    27, 20, 13,  6,  7, 14, 21, 28,
    35, 42, 49, 56, 57, 50, 43, 36,
    29, 22, 15, 23, 30, 37, 44, 51,
    58, 59, 52, 45, 38, 31, 39, 46,
    53, 60, 61, 54, 47, 55, 62, 63,
    // let corrupt input sample past end
    63, 63, 63, 63, 63, 63, 63, 63,
    63, 63, 63, 63, 63, 63, 63
};

// decode one 64-entry block--
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)
{
    int diff, dc, k;
    int t;

    if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
    t = stbi__jpeg_huff_decode(j, hdc);
    if (t < 0) return stbi__err("bad huffman code", "Corrupt JPEG");

    // 0 all the ac values now so we can do it 32-bits at a time
    memset(data, 0, 64 * sizeof(data[0]));

    diff = t ? stbi__extend_receive(j, t) : 0;
    dc = j->img_comp[b].dc_pred + diff;
    j->img_comp[b].dc_pred = dc;
    data[0] = (short)(dc * dequant[0]);

    // decode AC components, see JPEG spec
    k = 1;
    do {
        unsigned int zig;
        int c, r, s;
        if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
        c = (j->code_buffer » (32 - FAST_BITS)) & ((1 « FAST_BITS) - 1);
        r = fac[c];
        if (r) { // fast-AC path
            k += (r » 4) & 15; // run
            s = r & 15; // combined length
            j->code_buffer «= s;
            j->code_bits -= s;
            // decode into unzigzag'd location
            zig = stbi__jpeg_dezigzag[k++];
            data[zig] = (short)((r » 8) * dequant[zig]);
        }
        else {
            int rs = stbi__jpeg_huff_decode(j, hac);
            if (rs < 0) return stbi__err("bad huffman code", "Corrupt JPEG");
            s = rs & 15;
            r = rs » 4;
            if (s == 0) {
                if (rs != 0xf0) break; // end block
                k += 16;
            }
            else {
                k += r;
                // decode into unzigzag'd location
                zig = stbi__jpeg_dezigzag[k++];
                data[zig] = (short)(stbi__extend_receive(j, s) * dequant[zig]);
            }
        }
    } while (k < 64);
    return 1;
}

static int stbi__jpeg_decode_block_prog_dc(stbi__jpeg *j, short data[64], stbi__huffman *hdc, int b)
{
    int diff, dc;
    int t;
    if (j->spec_end != 0) return stbi__err("can't merge dc and ac", "Corrupt JPEG");

    if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);

    if (j->succ_high == 0) {
        // first scan for DC coefficient, must be first
        memset(data, 0, 64 * sizeof(data[0])); // 0 all the ac values now
        t = stbi__jpeg_huff_decode(j, hdc);
        diff = t ? stbi__extend_receive(j, t) : 0;

        dc = j->img_comp[b].dc_pred + diff;
        j->img_comp[b].dc_pred = dc;
        data[0] = (short)(dc « j->succ_low);
    }
    else {
        // refinement scan for DC coefficient
        if (stbi__jpeg_get_bit(j))
            data[0] += (short)(1 « j->succ_low);
    }
    return 1;
}

// @OPTIMIZE: store non-zigzagged during the decode passes,
// and only de-zigzag when dequantizing
static int stbi__jpeg_decode_block_prog_ac(stbi__jpeg *j, short data[64], stbi__huffman *hac,
  stbi__int16 *fac)
{
    int k;
    if (j->spec_start == 0) return stbi__err("can't merge dc and ac", "Corrupt JPEG");

    if (j->succ_high == 0) {
        int shift = j->succ_low;

        if (j->eob_run) {
            --j->eob_run;
            return 1;
        }

        k = j->spec_start;
        do {
            unsigned int zig;
            int c, r, s;
            if (j->code_bits < 16) stbi__grow_buffer_unsafe(j);
            c = (j->code_buffer » (32 - FAST_BITS)) & ((1 « FAST_BITS) - 1);
            r = fac[c];
            if (r) { // fast-AC path
                k += (r » 4) & 15; // run
                s = r & 15; // combined length
                j->code_buffer «= s;
                j->code_bits -= s;
                zig = stbi__jpeg_dezigzag[k++];
                data[zig] = (short)((r » 8) « shift);
            }
            else {
                int rs = stbi__jpeg_huff_decode(j, hac);
                if (rs < 0) return stbi__err("bad huffman code", "Corrupt JPEG");
                s = rs & 15;
                r = rs » 4;
                if (s == 0) {
                    if (r < 15) {
                        j->eob_run = (1 « r);
                        if (r)
                            j->eob_run += stbi__jpeg_get_bits(j, r);
                        --j->eob_run;
                        break;
                    }
                    k += 16;
                }
                else {
                    k += r;
                    zig = stbi__jpeg_dezigzag[k++];
                    data[zig] = (short)(stbi__extend_receive(j, s) « shift);
                }
            }
        } while (k <= j->spec_end);
    }
    else {
        // refinement scan for these AC coefficients

        short bit = (short)(1 « j->succ_low);

        if (j->eob_run) {
            --j->eob_run;
            for (k = j->spec_start; k <= j->spec_end; ++k) {
                short *p = &data[stbi__jpeg_dezigzag[k]];
                if (*p != 0)
                    if (stbi__jpeg_get_bit(j))
                        if ((*p & bit) == 0) {
                            if (*p > 0)
                                *p += bit;
                            else
                                *p -= bit;
                        }
            }
        }
        else {
            k = j->spec_start;
            do {
                int r, s;
                int rs = stbi__jpeg_huff_decode(j, hac); // @OPTIMIZE see if we can use the fast path
   here, advance-by-r is so slow, eh
                if (rs < 0) return stbi__err("bad huffman code", "Corrupt JPEG");
                s = rs & 15;
                r = rs » 4;
                if (s == 0) {
                    if (r < 15) {
                        j->eob_run = (1 « r) - 1;
                        if (r)
                            j->eob_run += stbi__jpeg_get_bits(j, r);
                        r = 64; // force end of block
                    }
                    else {
                        // r=15 s=0 should write 16 0s, so we just do
                        // a run of 15 0s and then write s (which is 0),
                        // so we don't have to do anything special here
                    }
                }
                else {
                    if (s != 1) return stbi__err("bad huffman code", "Corrupt JPEG");
                    // sign bit
                    if (stbi__jpeg_get_bit(j))
                        s = bit;
                    else
                        s = -bit;
                }

                // advance by r
                while (k <= j->spec_end) {
                    short *p = &data[stbi__jpeg_dezigzag[k++]];
                    if (*p != 0) {
                        if (stbi__jpeg_get_bit(j))
                            if ((*p & bit) == 0) {
                                if (*p > 0)
                                    *p += bit;
                                else
                                    *p -= bit;
                            }
                    }
                    else {
                        if (r == 0) {
                            *p = (short)s;
                            break;
                        }
                        --r;
                    }
                }
            } while (k <= j->spec_end);
        }
    }
    return 1;
}

// take a -128..127 value and stbi__clamp it and convert to 0..255
stbi_inline static stbi_uc stbi__clamp(int x)
{
    // trick to use a single test to catch both cases
    if ((unsigned int)x > 255) {
        if (x < 0) return 0;
        if (x > 255) return 255;
    }
    return (stbi_uc)x;
}

#define stbi__f2f(x)  ((int) (((x) * 4096 + 0.5)))
#define stbi__fsh(x)  ((x) « 12)

// derived from jidctint -- DCT_ISLOW
#define STBI__IDCT_1D(s0,s1,s2,s3,s4,s5,s6,s7) \
   int t0,t1,t2,t3,p1,p2,p3,p4,p5,x0,x1,x2,x3; \
   p2 = s2;                                    \
   p3 = s6;                                    \
   p1 = (p2+p3) * stbi__f2f(0.5411961f);       \
   t2 = p1 + p3*stbi__f2f(-1.847759065f);      \
   t3 = p1 + p2*stbi__f2f( 0.765366865f);      \
   p2 = s0;                                    \
   p3 = s4;                                    \
   t0 = stbi__fsh(p2+p3);                      \
   t1 = stbi__fsh(p2-p3);                      \
   x0 = t0+t3;                                 \
   x3 = t0-t3;                                 \
   x1 = t1+t2;                                 \
   x2 = t1-t2;                                 \
   t0 = s7;                                    \
   t1 = s5;                                    \
   t2 = s3;                                    \
   t3 = s1;                                    \
   p3 = t0+t2;                                 \
   p4 = t1+t3;                                 \
   p1 = t0+t3;                                 \
   p2 = t1+t2;                                 \
   p5 = (p3+p4)*stbi__f2f( 1.175875602f);      \
   t0 = t0*stbi__f2f( 0.298631336f);           \
   t1 = t1*stbi__f2f( 2.053119869f);           \
   t2 = t2*stbi__f2f( 3.072711026f);           \
   t3 = t3*stbi__f2f( 1.501321110f);           \
   p1 = p5 + p1*stbi__f2f(-0.899976223f);      \
   p2 = p5 + p2*stbi__f2f(-2.562915447f);      \
   p3 = p3*stbi__f2f(-1.961570560f);           \
   p4 = p4*stbi__f2f(-0.390180644f);           \
   t3 += p1+p4;                                \
   t2 += p2+p3;                                \
   t1 += p2+p4;                                \
   t0 += p1+p3;

static void stbi__idct_block(stbi_uc *out, int out_stride, short data[64])
{
    int i, val[64], *v = val;
    stbi_uc *o;
    short *d = data;

    // columns
    for (i = 0; i < 8; ++i, ++d, ++v) {
        // if all zeroes, shortcut -- this avoids dequantizing 0s and IDCTing
        if (d[8] == 0 && d[16] == 0 && d[24] == 0 && d[32] == 0
            && d[40] == 0 && d[48] == 0 && d[56] == 0) {
            //    no shortcut                 0     seconds
            //    (1|2|3|4|5|6|7)==0          0     seconds
            //    all separate               -0.047 seconds
            //    1 && 2|3 && 4|5 && 6|7:    -0.047 seconds
            int dcterm = d[0] « 2;
            v[0] = v[8] = v[16] = v[24] = v[32] = v[40] = v[48] = v[56] = dcterm;
        }
        else {
            STBI__IDCT_1D(d[0], d[8], d[16], d[24], d[32], d[40], d[48], d[56])
                // constants scaled things up by 1«12; let's bring them back
                // down, but keep 2 extra bits of precision
                x0 += 512; x1 += 512; x2 += 512; x3 += 512;
            v[0] = (x0 + t3) » 10;
            v[56] = (x0 - t3) » 10;
            v[8] = (x1 + t2) » 10;
            v[48] = (x1 - t2) » 10;
            v[16] = (x2 + t1) » 10;
            v[40] = (x2 - t1) » 10;
            v[24] = (x3 + t0) » 10;
            v[32] = (x3 - t0) » 10;
        }
    }

    for (i = 0, v = val, o = out; i < 8; ++i, v += 8, o += out_stride) {
        // no fast case since the first 1D IDCT spread components out
        STBI__IDCT_1D(v[0], v[1], v[2], v[3], v[4], v[5], v[6], v[7])
            // constants scaled things up by 1«12, plus we had 1«2 from first
            // loop, plus horizontal and vertical each scale by sqrt(8) so together
            // we've got an extra 1«3, so 1«17 total we need to remove.
            // so we want to round that, which means adding 0.5 * 1«17,
            // aka 65536. Also, we'll end up with -128 to 127 that we want
            // to encode as 0..255 by adding 128, so we'll add that before the shift
            x0 += 65536 + (128 « 17);
        x1 += 65536 + (128 « 17);
        x2 += 65536 + (128 « 17);
        x3 += 65536 + (128 « 17);
        // tried computing the shifts into temps, or'ing the temps to see
        // if any were out of range, but that was slower
        o[0] = stbi__clamp((x0 + t3) » 17);
        o[7] = stbi__clamp((x0 - t3) » 17);
        o[1] = stbi__clamp((x1 + t2) » 17);
        o[6] = stbi__clamp((x1 - t2) » 17);
        o[2] = stbi__clamp((x2 + t1) » 17);
        o[5] = stbi__clamp((x2 - t1) » 17);
        o[3] = stbi__clamp((x3 + t0) » 17);
        o[4] = stbi__clamp((x3 - t0) » 17);
    }
}

#ifdef STBI_SSE2
// sse2 integer IDCT. not the fastest possible implementation but it
// produces bit-identical results to the generic C version so it's
// fully "transparent".
static void stbi__idct_simd(stbi_uc *out, int out_stride, short data[64])
{
    // This is constructed to match our regular (generic) integer IDCT exactly.
    __m128i row0, row1, row2, row3, row4, row5, row6, row7;
    __m128i tmp;

    // dot product constant: even elems=x, odd elems=y
#define dct_const(x,y)  _mm_setr_epi16((x),(y),(x),(y),(x),(y),(x),(y))

    // out(0) = c0[even]*x + c0[odd]*y   (c0, x, y 16-bit, out 32-bit)
    // out(1) = c1[even]*x + c1[odd]*y
#define dct_rot(out0,out1, x,y,c0,c1) \
      __m128i c0##lo = _mm_unpacklo_epi16((x),(y)); \
      __m128i c0##hi = _mm_unpackhi_epi16((x),(y)); \
      __m128i out0##_l = _mm_madd_epi16(c0##lo, c0); \
      __m128i out0##_h = _mm_madd_epi16(c0##hi, c0); \
      __m128i out1##_l = _mm_madd_epi16(c0##lo, c1); \
      __m128i out1##_h = _mm_madd_epi16(c0##hi, c1)

    // out = in « 12  (in 16-bit, out 32-bit)
#define dct_widen(out, in) \
      __m128i out##_l = _mm_srai_epi32(_mm_unpacklo_epi16(_mm_setzero_si128(), (in)), 4); \
      __m128i out##_h = _mm_srai_epi32(_mm_unpackhi_epi16(_mm_setzero_si128(), (in)), 4)

    // wide add
#define dct_wadd(out, a, b) \
      __m128i out##_l = _mm_add_epi32(a##_l, b##_l); \
      __m128i out##_h = _mm_add_epi32(a##_h, b##_h)

    // wide sub
#define dct_wsub(out, a, b) \
      __m128i out##_l = _mm_sub_epi32(a##_l, b##_l); \
      __m128i out##_h = _mm_sub_epi32(a##_h, b##_h)

    // butterfly a/b, add bias, then shift by "s" and pack
#define dct_bfly32o(out0, out1, a,b,bias,s) \
      { \
         __m128i abiased_l = _mm_add_epi32(a##_l, bias); \
         __m128i abiased_h = _mm_add_epi32(a##_h, bias); \
         dct_wadd(sum, abiased, b); \
         dct_wsub(dif, abiased, b); \
         out0 = _mm_packs_epi32(_mm_srai_epi32(sum_l, s), _mm_srai_epi32(sum_h, s)); \
         out1 = _mm_packs_epi32(_mm_srai_epi32(dif_l, s), _mm_srai_epi32(dif_h, s)); \
      }

    // 8-bit interleave step (for transposes)
#define dct_interleave8(a, b) \
      tmp = a; \
      a = _mm_unpacklo_epi8(a, b); \
      b = _mm_unpackhi_epi8(tmp, b)

    // 16-bit interleave step (for transposes)
#define dct_interleave16(a, b) \
      tmp = a; \
      a = _mm_unpacklo_epi16(a, b); \
      b = _mm_unpackhi_epi16(tmp, b)

#define dct_pass(bias,shift) \
      { \
         /* even part */ \
         dct_rot(t2e,t3e, row2,row6, rot0_0,rot0_1); \
         __m128i sum04 = _mm_add_epi16(row0, row4); \
         __m128i dif04 = _mm_sub_epi16(row0, row4); \
         dct_widen(t0e, sum04); \
         dct_widen(t1e, dif04); \
         dct_wadd(x0, t0e, t3e); \
         dct_wsub(x3, t0e, t3e); \
         dct_wadd(x1, t1e, t2e); \
         dct_wsub(x2, t1e, t2e); \
         /* odd part */ \
         dct_rot(y0o,y2o, row7,row3, rot2_0,rot2_1); \
         dct_rot(y1o,y3o, row5,row1, rot3_0,rot3_1); \
         __m128i sum17 = _mm_add_epi16(row1, row7); \
         __m128i sum35 = _mm_add_epi16(row3, row5); \
         dct_rot(y4o,y5o, sum17,sum35, rot1_0,rot1_1); \
         dct_wadd(x4, y0o, y4o); \
         dct_wadd(x5, y1o, y5o); \
         dct_wadd(x6, y2o, y5o); \
         dct_wadd(x7, y3o, y4o); \
         dct_bfly32o(row0,row7, x0,x7,bias,shift); \
         dct_bfly32o(row1,row6, x1,x6,bias,shift); \
         dct_bfly32o(row2,row5, x2,x5,bias,shift); \
         dct_bfly32o(row3,row4, x3,x4,bias,shift); \
      }

   __m128i rot0_0 = dct_const(stbi__f2f(0.5411961f), stbi__f2f(0.5411961f) +
stbi__f2f(-1.847759065f));
   __m128i rot0_1 = dct_const(stbi__f2f(0.5411961f) + stbi__f2f(0.765366865f), stbi__f2f(0.5411961f));
   __m128i rot1_0 = dct_const(stbi__f2f(1.175875602f) + stbi__f2f(-0.899976223f),
stbi__f2f(1.175875602f));
   __m128i rot1_1 = dct_const(stbi__f2f(1.175875602f), stbi__f2f(1.175875602f) +
stbi__f2f(-2.562915447f));
   __m128i rot2_0 = dct_const(stbi__f2f(-1.961570560f) + stbi__f2f(0.298631336f),
stbi__f2f(-1.961570560f));
   __m128i rot2_1 = dct_const(stbi__f2f(-1.961570560f), stbi__f2f(-1.961570560f) +
stbi__f2f(3.072711026f));
   __m128i rot3_0 = dct_const(stbi__f2f(-0.390180644f) + stbi__f2f(2.053119869f),
stbi__f2f(-0.390180644f));
   __m128i rot3_1 = dct_const(stbi__f2f(-0.390180644f), stbi__f2f(-0.390180644f) +
stbi__f2f(1.501321110f));

   // rounding biases in column/row passes, see stbi__idct_block for explanation.
   __m128i bias_0 = _mm_set1_epi32(512);
   __m128i bias_1 = _mm_set1_epi32(65536 + (128 « 17));

   // load
   row0 = _mm_load_si128((const __m128i *) (data + 0 * 8));
   row1 = _mm_load_si128((const __m128i *) (data + 1 * 8));
   row2 = _mm_load_si128((const __m128i *) (data + 2 * 8));
   row3 = _mm_load_si128((const __m128i *) (data + 3 * 8));
   row4 = _mm_load_si128((const __m128i *) (data + 4 * 8));
   row5 = _mm_load_si128((const __m128i *) (data + 5 * 8));
   row6 = _mm_load_si128((const __m128i *) (data + 6 * 8));
   row7 = _mm_load_si128((const __m128i *) (data + 7 * 8));

   // column pass
   dct_pass(bias_0, 10);

   {
      // 16bit 8x8 transpose pass 1
      dct_interleave16(row0, row4);
      dct_interleave16(row1, row5);
      dct_interleave16(row2, row6);
      dct_interleave16(row3, row7);

      // transpose pass 2
      dct_interleave16(row0, row2);
      dct_interleave16(row1, row3);
      dct_interleave16(row4, row6);
      dct_interleave16(row5, row7);

      // transpose pass 3
      dct_interleave16(row0, row1);
      dct_interleave16(row2, row3);
      dct_interleave16(row4, row5);
      dct_interleave16(row6, row7);
   }

   // row pass
   dct_pass(bias_1, 17);

   {
      // pack
      __m128i p0 = _mm_packus_epi16(row0, row1); // a0a1a2a3...a7b0b1b2b3...b7
      __m128i p1 = _mm_packus_epi16(row2, row3);
      __m128i p2 = _mm_packus_epi16(row4, row5);
      __m128i p3 = _mm_packus_epi16(row6, row7);

      // 8bit 8x8 transpose pass 1
      dct_interleave8(p0, p2); // a0e0a1e1...
```

```
        dct_interleave8(p1, p3); // c0g0c1g1...

                                 // transpose pass 2
        dct_interleave8(p0, p1); // a0c0e0g0...
        dct_interleave8(p2, p3); // b0d0f0h0...

                                 // transpose pass 3
        dct_interleave8(p0, p2); // a0b0c0d0...
        dct_interleave8(p1, p3); // a4b4c4d4...

                                 // store
        _mm_storel_epi64((__m128i *) out, p0); out += out_stride;
        _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p0, 0x4e)); out += out_stride;
        _mm_storel_epi64((__m128i *) out, p2); out += out_stride;
        _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p2, 0x4e)); out += out_stride;
        _mm_storel_epi64((__m128i *) out, p1); out += out_stride;
        _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p1, 0x4e)); out += out_stride;
        _mm_storel_epi64((__m128i *) out, p3); out += out_stride;
        _mm_storel_epi64((__m128i *) out, _mm_shuffle_epi32(p3, 0x4e));
    }

#undef dct_const
#undef dct_rot
#undef dct_widen
#undef dct_wadd
#undef dct_wsub
#undef dct_bfly32o
#undef dct_interleave8
#undef dct_interleave16
#undef dct_pass
}

#endif // STBI_SSE2

#ifdef STBI_NEON

// NEON integer IDCT. should produce bit-identical
// results to the generic C version.
static void stbi__idct_simd(stbi_uc *out, int out_stride, short data[64])
{
    int16x8_t row0, row1, row2, row3, row4, row5, row6, row7;

    int16x4_t rot0_0 = vdup_n_s16(stbi__f2f(0.5411961f));
    int16x4_t rot0_1 = vdup_n_s16(stbi__f2f(-1.847759065f));
    int16x4_t rot0_2 = vdup_n_s16(stbi__f2f(0.765366865f));
    int16x4_t rot1_0 = vdup_n_s16(stbi__f2f(1.175875602f));
    int16x4_t rot1_1 = vdup_n_s16(stbi__f2f(-0.899976223f));
    int16x4_t rot1_2 = vdup_n_s16(stbi__f2f(-2.562915447f));
    int16x4_t rot2_0 = vdup_n_s16(stbi__f2f(-1.961570560f));
    int16x4_t rot2_1 = vdup_n_s16(stbi__f2f(-0.390180644f));
    int16x4_t rot3_0 = vdup_n_s16(stbi__f2f(0.298631336f));
    int16x4_t rot3_1 = vdup_n_s16(stbi__f2f(2.053119869f));
    int16x4_t rot3_2 = vdup_n_s16(stbi__f2f(3.072711026f));
    int16x4_t rot3_3 = vdup_n_s16(stbi__f2f(1.501321110f));

#define dct_long_mul(out, inq, coeff) \
   int32x4_t out##_l = vmull_s16(vget_low_s16(inq), coeff); \
   int32x4_t out##_h = vmull_s16(vget_high_s16(inq), coeff)

#define dct_long_mac(out, acc, inq, coeff) \
   int32x4_t out##_l = vmlal_s16(acc##_l, vget_low_s16(inq), coeff); \
   int32x4_t out##_h = vmlal_s16(acc##_h, vget_high_s16(inq), coeff)

#define dct_widen(out, inq) \
   int32x4_t out##_l = vshll_n_s16(vget_low_s16(inq), 12); \
   int32x4_t out##_h = vshll_n_s16(vget_high_s16(inq), 12)

    // wide add
#define dct_wadd(out, a, b) \
   int32x4_t out##_l = vaddq_s32(a##_l, b##_l); \
   int32x4_t out##_h = vaddq_s32(a##_h, b##_h)

    // wide sub
#define dct_wsub(out, a, b) \
   int32x4_t out##_l = vsubq_s32(a##_l, b##_l); \
   int32x4_t out##_h = vsubq_s32(a##_h, b##_h)

    // butterfly a/b, then shift using "shiftop" by "s" and pack
#define dct_bfly32o(out0,out1, a,b,shiftop,s) \
   { \
      dct_wadd(sum, a, b); \
      dct_wsub(dif, a, b); \
      out0 = vcombine_s16(shiftop(sum_l, s), shiftop(sum_h, s)); \
      out1 = vcombine_s16(shiftop(dif_l, s), shiftop(dif_h, s)); \
   }

#define dct_pass(shiftop, shift) \
   { \
      /* even part */ \
      int16x8_t sum26 = vaddq_s16(row2, row6); \
      dct_long_mul(p1e, sum26, rot0_0); \
      dct_long_mac(t2e, p1e, row6, rot0_1); \
      dct_long_mac(t3e, p1e, row2, rot0_2); \
      int16x8_t sum04 = vaddq_s16(row0, row4); \
      int16x8_t dif04 = vsubq_s16(row0, row4); \
      dct_widen(t0e, sum04); \
      dct_widen(t1e, dif04); \
      dct_wadd(x0, t0e, t3e); \
      dct_wsub(x3, t0e, t3e); \
      dct_wadd(x1, t1e, t2e); \
      dct_wsub(x2, t1e, t2e); \
      /* odd part */ \
      int16x8_t sum15 = vaddq_s16(row1, row5); \
      int16x8_t sum17 = vaddq_s16(row1, row7); \
      int16x8_t sum35 = vaddq_s16(row3, row5); \
      int16x8_t sum37 = vaddq_s16(row3, row7); \
      int16x8_t sumodd = vaddq_s16(sum17, sum35); \
      dct_long_mul(p5o, sumodd, rot1_0); \
      dct_long_mac(p1o, p5o, sum17, rot1_1); \
      dct_long_mac(p2o, p5o, sum35, rot1_2); \
      dct_long_mul(p3o, sum37, rot2_0); \
      dct_long_mul(p4o, sum15, rot2_1); \
      dct_wadd(sump13o, p1o, p3o); \
      dct_wadd(sump24o, p2o, p4o); \
      dct_wadd(sump23o, p2o, p3o); \
      dct_wadd(sump14o, p1o, p4o); \
      dct_long_mac(x4, sump13o, row7, rot3_0); \
      dct_long_mac(x5, sump24o, row5, rot3_1); \
      dct_long_mac(x6, sump23o, row3, rot3_2); \
      dct_long_mac(x7, sump14o, row1, rot3_3); \
      dct_bfly32o(row0,row7, x0,x7,shiftop,shift); \
      dct_bfly32o(row1,row6, x1,x6,shiftop,shift); \
      dct_bfly32o(row2,row5, x2,x5,shiftop,shift); \
      dct_bfly32o(row3,row4, x3,x4,shiftop,shift); \
   }

    // load
    row0 = vld1q_s16(data + 0 * 8);
    row1 = vld1q_s16(data + 1 * 8);
    row2 = vld1q_s16(data + 2 * 8);
    row3 = vld1q_s16(data + 3 * 8);
    row4 = vld1q_s16(data + 4 * 8);
    row5 = vld1q_s16(data + 5 * 8);
    row6 = vld1q_s16(data + 6 * 8);
    row7 = vld1q_s16(data + 7 * 8);

    // add DC bias
    row0 = vaddq_s16(row0, vsetq_lane_s16(1024, vdupq_n_s16(0), 0));

    // column pass
    dct_pass(vrshrn_n_s32, 10);

    // 16bit 8x8 transpose
    {
        // these three map to a single VTRN.16, VTRN.32, and VSWP, respectively.
        // whether compilers actually get this is another story, sadly.
#define dct_trn16(x, y) { int16x8x2_t t = vtrnq_s16(x, y); x = t.val[0]; y = t.val[1]; }
#define dct_trn32(x, y) { int32x4x2_t t = vtrnq_s32(vreinterpretq_s32_s16(x),
  vreinterpretq_s32_s16(y)); x = vreinterpretq_s16_s32(t.val[0]); y = vreinterpretq_s16_s32(t.val[1]);
  }
#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)); }

        // pass 1
        dct_trn16(row0, row1); // a0b0a2b2a4b4a6b6
        dct_trn16(row2, row3);
        dct_trn16(row4, row5);
        dct_trn16(row6, row7);

        // pass 2
        dct_trn32(row0, row2); // a0b0c0d0a4b4c4d4
        dct_trn32(row1, row3);
        dct_trn32(row4, row6);
        dct_trn32(row5, row7);

        // pass 3
        dct_trn64(row0, row4); // a0b0c0d0e0f0g0h0
        dct_trn64(row1, row5);
        dct_trn64(row2, row6);
        dct_trn64(row3, row7);

#undef dct_trn16
#undef dct_trn32
#undef dct_trn64
```

```
    }

    // row pass
    // vrshrn_n_s32 only supports shifts up to 16, we need
    // 17. so do a non-rounding shift of 16 first then follow
    // up with a rounding shift by 1.
    dct_pass(vshrn_n_s32, 16);

    {
        // pack and round
        uint8x8_t p0 = vqrshrun_n_s16(row0, 1);
        uint8x8_t p1 = vqrshrun_n_s16(row1, 1);
        uint8x8_t p2 = vqrshrun_n_s16(row2, 1);
        uint8x8_t p3 = vqrshrun_n_s16(row3, 1);
        uint8x8_t p4 = vqrshrun_n_s16(row4, 1);
        uint8x8_t p5 = vqrshrun_n_s16(row5, 1);
        uint8x8_t p6 = vqrshrun_n_s16(row6, 1);
        uint8x8_t p7 = vqrshrun_n_s16(row7, 1);

        // again, these can translate into one instruction, but often don't.
#define dct_trn8_8(x, y) { uint8x8x2_t t = vtrn_u8(x, y); x = t.val[0]; y = t.val[1]; }
#define dct_trn8_16(x, y) { uint16x4x2_t t = vtrn_u16(vreinterpret_u16_u8(x), vreinterpret_u16_u8(y));
  x = vreinterpret_u8_u16(t.val[0]); y = vreinterpret_u8_u16(t.val[1]); }
#define dct_trn8_32(x, y) { uint32x2x2_t t = vtrn_u32(vreinterpret_u32_u8(x), vreinterpret_u32_u8(y));
  x = vreinterpret_u8_u32(t.val[0]); y = vreinterpret_u8_u32(t.val[1]); }

        // sadly can't use interleaved stores here since we only write
        // 8 bytes to each scan line!

        // 8x8 8-bit transpose pass 1
        dct_trn8_8(p0, p1);
        dct_trn8_8(p2, p3);
        dct_trn8_8(p4, p5);
        dct_trn8_8(p6, p7);

        // pass 2
        dct_trn8_16(p0, p2);
        dct_trn8_16(p1, p3);
        dct_trn8_16(p4, p6);
        dct_trn8_16(p5, p7);

        // pass 3
        dct_trn8_32(p0, p4);
        dct_trn8_32(p1, p5);
        dct_trn8_32(p2, p6);
        dct_trn8_32(p3, p7);

        // store
        vst1_u8(out, p0); out += out_stride;
        vst1_u8(out, p1); out += out_stride;
        vst1_u8(out, p2); out += out_stride;
        vst1_u8(out, p3); out += out_stride;
        vst1_u8(out, p4); out += out_stride;
        vst1_u8(out, p5); out += out_stride;
        vst1_u8(out, p6); out += out_stride;
        vst1_u8(out, p7);

#undef dct_trn8_8
#undef dct_trn8_16
#undef dct_trn8_32
    }

#undef dct_long_mul
#undef dct_long_mac
#undef dct_widen
#undef dct_wadd
#undef dct_wsub
#undef dct_bfly32o
#undef dct_pass
}

#endif // STBI_NEON

#define STBI__MARKER_none  0xff
// if there's a pending marker from the entropy stream, return that
// otherwise, fetch from the stream and get a marker. if there's no
// marker, return 0xff, which is never a valid marker value
static stbi_uc stbi__get_marker(stbi__jpeg *j)
{
    stbi_uc x;
    if (j->marker != STBI__MARKER_none) { x = j->marker; j->marker = STBI__MARKER_none; return x; }
    x = stbi__get8(j->s);
    if (x != 0xff) return STBI__MARKER_none;
    while (x == 0xff)
        x = stbi__get8(j->s);
    return x;
}
```

```

// in each scan, we'll have scan_n components, and the order
// of the components is specified by order[]
#define STBI__RESTART(x)     ((x) >= 0xd0 && (x) <= 0xd7)

// after a restart interval, stbi__jpeg_reset the entropy decoder and
// the dc prediction
static void stbi__jpeg_reset(stbi__jpeg *j)
{
    j->code_bits = 0;
    j->code_buffer = 0;
    j->nomore = 0;
    j->img_comp[0].dc_pred = j->img_comp[1].dc_pred = j->img_comp[2].dc_pred = 0;
    j->marker = STBI__MARKER_none;
    j->todo = j->restart_interval ? j->restart_interval : 0x7fffffff;
    j->eob_run = 0;
    // no more than 1«31 MCUs if no restart_interal? that's plenty safe,
    // since we don't even allow 1«30 pixels
}

static int stbi__parse_entropy_coded_data(stbi__jpeg *z)
{
    stbi__jpeg_reset(z);
    if (!z->progressive) {
        if (z->scan_n == 1) {
            int i, j;
            STBI_SIMD_ALIGN(short, data[64]);
            int n = z->order[0];
            // non-interleaved data, we just need to process one block at a time,
            // in trivial scanline order
            // number of blocks to do just depends on how many actual "pixels" this
            // component has, independent of interleaved MCU blocking and such
            int w = (z->img_comp[n].x + 7) » 3;
            int h = (z->img_comp[n].y + 7) » 3;
            for (j = 0; j < h; ++j) {
                for (i = 0; i < w; ++i) {
                    int ha = z->img_comp[n].ha;
                    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;
                    z->idct_block_kernel(z->img_comp[n].data + z->img_comp[n].w2*j * 8 + i * 8,
   z->img_comp[n].w2, data);
                    // every data block is an MCU, so countdown the restart interval
                    if (--z->todo <= 0) {
                        if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
                        // if it's NOT a restart, then just bail, so we get corrupt data
                        // rather than no data
                        if (!STBI__RESTART(z->marker)) return 1;
                        stbi__jpeg_reset(z);
                    }
                }
            }
            return 1;
        }
        else { // interleaved
            int i, j, k, x, y;
            STBI_SIMD_ALIGN(short, data[64]);
            for (j = 0; j < z->img_mcu_y; ++j) {
                for (i = 0; i < z->img_mcu_x; ++i) {
                    // scan an interleaved mcu... process scan_n components in order
                    for (k = 0; k < z->scan_n; ++k) {
                        int n = z->order[k];
                        // scan out an mcu's worth of this component; that's just determined
                        // by the basic H and V specified for the component
                        for (y = 0; y < z->img_comp[n].v; ++y) {
                            for (x = 0; x < z->img_comp[n].h; ++x) {
                                int x2 = (i*z->img_comp[n].h + x) * 8;
                                int y2 = (j*z->img_comp[n].v + y) * 8;
                                int ha = z->img_comp[n].ha;
                                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;
                                z->idct_block_kernel(z->img_comp[n].data + z->img_comp[n].w2*y2 + x2,
   z->img_comp[n].w2, data);
                            }
                        }
                    }
                    // after all interleaved components, that's an interleaved MCU,
                    // so now count down the restart interval
                    if (--z->todo <= 0) {
                        if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
                        if (!STBI__RESTART(z->marker)) return 1;
                        stbi__jpeg_reset(z);
                    }
                }
            }
            return 1;
        }
    }
    else {
        if (z->scan_n == 1) {
            int i, j;
            int n = z->order[0];
            // non-interleaved data, we just need to process one block at a time,
            // in trivial scanline order
            // number of blocks to do just depends on how many actual "pixels" this
            // component has, independent of interleaved MCU blocking and such
            int w = (z->img_comp[n].x + 7) » 3;
            int h = (z->img_comp[n].y + 7) » 3;
            for (j = 0; j < h; ++j) {
                for (i = 0; i < w; ++i) {
                    short *data = z->img_comp[n].coeff + 64 * (i + j * z->img_comp[n].coeff_w);
                    if (z->spec_start == 0) {
                        if (!stbi__jpeg_decode_block_prog_dc(z, data, &z->huff_dc[z->img_comp[n].hd],
 n))
                            return 0;
                    }
                    else {
                        int ha = z->img_comp[n].ha;
                        if (!stbi__jpeg_decode_block_prog_ac(z, data, &z->huff_ac[ha], z->fast_ac[ha]))
                            return 0;
                    }
                    // every data block is an MCU, so countdown the restart interval
                    if (--z->todo <= 0) {
                        if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
                        if (!STBI__RESTART(z->marker)) return 1;
                        stbi__jpeg_reset(z);
                    }
                }
            }
            return 1;
        }
        else { // interleaved
            int i, j, k, x, y;
            for (j = 0; j < z->img_mcu_y; ++j) {
                for (i = 0; i < z->img_mcu_x; ++i) {
                    // scan an interleaved mcu... process scan_n components in order
                    for (k = 0; k < z->scan_n; ++k) {
                        int n = z->order[k];
                        // scan out an mcu's worth of this component; that's just determined
                        // by the basic H and V specified for the component
                        for (y = 0; y < z->img_comp[n].v; ++y) {
                            for (x = 0; x < z->img_comp[n].h; ++x) {
                                int x2 = (i*z->img_comp[n].h + x);
                                int y2 = (j*z->img_comp[n].v + y);
                                short *data = z->img_comp[n].coeff + 64 * (x2 + y2 *
 z->img_comp[n].coeff_w);
                                if (!stbi__jpeg_decode_block_prog_dc(z, data,
 &z->huff_dc[z->img_comp[n].hd], n))
                                    return 0;
                            }
                        }
                    }
                    // after all interleaved components, that's an interleaved MCU,
                    // so now count down the restart interval
                    if (--z->todo <= 0) {
                        if (z->code_bits < 24) stbi__grow_buffer_unsafe(z);
                        if (!STBI__RESTART(z->marker)) return 1;
                        stbi__jpeg_reset(z);
                    }
                }
            }
            return 1;
        }
    }
}

static void stbi__jpeg_dequantize(short *data, stbi_uc *dequant)
{
    int i;
    for (i = 0; i < 64; ++i)
        data[i] *= dequant[i];
}

static void stbi__jpeg_finish(stbi__jpeg *z)
{
    if (z->progressive) {
        // dequantize and idct the data
        int i, j, n;
        for (n = 0; n < z->s->img_n; ++n) {
            int w = (z->img_comp[n].x + 7) » 3;
            int h = (z->img_comp[n].y + 7) » 3;
            for (j = 0; j < h; ++j) {
                for (i = 0; i < w; ++i) {
                    short *data = z->img_comp[n].coeff + 64 * (i + j * z->img_comp[n].coeff_w);
                    stbi__jpeg_dequantize(data, z->dequant[z->img_comp[n].tq]);
                    z->idct_block_kernel(z->img_comp[n].data + z->img_comp[n].w2*j * 8 + i * 8,
   z->img_comp[n].w2, data);
                }
            }
        }
    }
}

static int stbi__process_marker(stbi__jpeg *z, int m)
{
    int L;
    switch (m) {
    case STBI__MARKER_none: // no marker found
        return stbi__err("expected marker", "Corrupt JPEG");

    case 0xDD: // DRI - specify restart interval
        if (stbi__get16be(z->s) != 4) return stbi__err("bad DRI len", "Corrupt JPEG");
        z->restart_interval = stbi__get16be(z->s);
        return 1;

    case 0xDB: // DQT - define quantization table
        L = stbi__get16be(z->s) - 2;
        while (L > 0) {
            int q = stbi__get8(z->s);
            int p = q » 4;
            int t = q & 15, i;
            if (p != 0) return stbi__err("bad DQT type", "Corrupt JPEG");
            if (t > 3) return stbi__err("bad DQT table", "Corrupt JPEG");
            for (i = 0; i < 64; ++i)
                z->dequant[t][stbi__jpeg_dezigzag[i]] = stbi__get8(z->s);
            L -= 65;
        }
        return L == 0;

    case 0xC4: // DHT - define huffman table
        L = stbi__get16be(z->s) - 2;
        while (L > 0) {
            stbi_uc *v;
            int sizes[16], i, n = 0;
            int q = stbi__get8(z->s);
            int tc = q » 4;
            int th = q & 15;
            if (tc > 1 || th > 3) return stbi__err("bad DHT header", "Corrupt JPEG");
            for (i = 0; i < 16; ++i) {
                sizes[i] = stbi__get8(z->s);
                n += sizes[i];
            }
            L -= 17;
            if (tc == 0) {
                if (!stbi__build_huffman(z->huff_dc + th, sizes)) return 0;
                v = z->huff_dc[th].values;
            }
            else {
                if (!stbi__build_huffman(z->huff_ac + th, sizes)) return 0;
                v = z->huff_ac[th].values;
            }
            for (i = 0; i < n; ++i)
                v[i] = stbi__get8(z->s);
            if (tc != 0)
                stbi__build_fast_ac(z->fast_ac[th], z->huff_ac + th);
            L -= n;
        }
        return L == 0;
    }
    // check for comment block or APP blocks
    if ((m >= 0xE0 && m <= 0xEF) || m == 0xFE) {
        stbi__skip(z->s, stbi__get16be(z->s) - 2);
        return 1;
    }
    return 0;
}

// after we see SOS
static int stbi__process_scan_header(stbi__jpeg *z)
{
    int i;
    int Ls = stbi__get16be(z->s);
    z->scan_n = stbi__get8(z->s);
    if (z->scan_n < 1 || z->scan_n > 4 || z->scan_n > (int)z->s->img_n) return stbi__err("bad SOS
   component count", "Corrupt JPEG");
    if (Ls != 6 + 2 * z->scan_n) return stbi__err("bad SOS len", "Corrupt JPEG");
    for (i = 0; i < z->scan_n; ++i) {
        int id = stbi__get8(z->s), which;
        int q = stbi__get8(z->s);
        for (which = 0; which < z->s->img_n; ++which)
            if (z->img_comp[which].id == id)
                break;
        if (which == z->s->img_n) return 0; // no match
        z->img_comp[which].hd = q » 4;   if (z->img_comp[which].hd > 3) return stbi__err("bad DC huff", "Corrupt JPEG");
        z->img_comp[which].ha = q & 15;   if (z->img_comp[which].ha > 3) return stbi__err("bad AC huff", "Corrupt JPEG");
        z->order[i] = which;
    }

    {
        int aa;
        z->spec_start = stbi__get8(z->s);
        z->spec_end = stbi__get8(z->s); // should be 63, but might be 0
        aa = stbi__get8(z->s);
        z->succ_high = (aa » 4);
        z->succ_low = (aa & 15);
        if (z->progressive) {
            if (z->spec_start > 63 || z->spec_end > 63 || z->spec_start > z->spec_end || z->succ_high > 13 || z->succ_low > 13)
                return stbi__err("bad SOS", "Corrupt JPEG");
        }
        else {
            if (z->spec_start != 0) return stbi__err("bad SOS", "Corrupt JPEG");
            if (z->succ_high != 0 || z->succ_low != 0) return stbi__err("bad SOS", "Corrupt JPEG");
            z->spec_end = 63;
        }
    }

    return 1;
}

static int stbi__free_jpeg_components(stbi__jpeg *z, int ncomp, int why)
{
    int i;
    for (i = 0; i < ncomp; ++i) {
        if (z->img_comp[i].raw_data) {
            STBI_FREE(z->img_comp[i].raw_data);
            z->img_comp[i].raw_data = NULL;
            z->img_comp[i].data = NULL;
        }
        if (z->img_comp[i].raw_coeff) {
            STBI_FREE(z->img_comp[i].raw_coeff);
            z->img_comp[i].raw_coeff = 0;
            z->img_comp[i].coeff = 0;
        }
        if (z->img_comp[i].linebuf) {
            STBI_FREE(z->img_comp[i].linebuf);
            z->img_comp[i].linebuf = NULL;
        }
    }
    return why;
}

static int stbi__process_frame_header(stbi__jpeg *z, int scan)
{
    stbi__context *s = z->s;
    int Lf, p, i, q, h_max = 1, v_max = 1, c;
    Lf = stbi__get16be(s);          if (Lf < 11) return stbi__err("bad SOF len", "Corrupt JPEG"); // JPEG
    p = stbi__get8(s);             if (p != 8) return stbi__err("only 8-bit", "JPEG format not supported: 8-bit only"); // JPEG baseline
    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
    s->img_x = stbi__get16be(s);   if (s->img_x == 0) return stbi__err("0 width", "Corrupt JPEG"); // JPEG requires
    c = stbi__get8(s);
    if (c != 3 && c != 1) return stbi__err("bad component count", "Corrupt JPEG");    // JFIF requires
    s->img_n = c;
    for (i = 0; i < c; ++i) {
        z->img_comp[i].data = NULL;
        z->img_comp[i].linebuf = NULL;
    }

    if (Lf != 8 + 3 * s->img_n) return stbi__err("bad SOF len", "Corrupt JPEG");

    z->rgb = 0;
    for (i = 0; i < s->img_n; ++i) {
        static unsigned char rgb[3] = { 'R', 'G', 'B' };
        z->img_comp[i].id = stbi__get8(s);
        if (z->img_comp[i].id != i + 1)   // JFIF requires
            if (z->img_comp[i].id != i) {  // some version of jpegtran outputs non-JFIF-compliant files!
                                           // somethings output this (see http://fileformats.archiveteam.org/wiki/JPEG#Color_format)
                if (z->img_comp[i].id != rgb[i])
                    return stbi__err("bad component ID", "Corrupt JPEG");
                ++z->rgb;
            }
        q = stbi__get8(s);
        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");
        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");
        z->img_comp[i].tq = stbi__get8(s);  if (z->img_comp[i].tq > 3) return stbi__err("bad TQ", "Corrupt JPEG");
    }

    if (scan != STBI__SCAN_load) return 1;

    if (!stbi__mad3sizes_valid(s->img_x, s->img_y, s->img_n, 0)) return stbi__err("too large", "Image too large to decode");

    for (i = 0; i < s->img_n; ++i) {
        if (z->img_comp[i].h > h_max) h_max = z->img_comp[i].h;
        if (z->img_comp[i].v > v_max) v_max = z->img_comp[i].v;
    }

    // compute interleaved mcu info
    z->img_h_max = h_max;
    z->img_v_max = v_max;
    z->img_mcu_w = h_max * 8;
    z->img_mcu_h = v_max * 8;
    // these sizes can’t be more than 17 bits
    z->img_mcu_x = (s->img_x + z->img_mcu_w - 1) / z->img_mcu_w;
    z->img_mcu_y = (s->img_y + z->img_mcu_h - 1) / z->img_mcu_h;

    for (i = 0; i < s->img_n; ++i) {
        // number of effective pixels (e.g. for non-interleaved MCU)
        z->img_comp[i].x = (s->img_x * z->img_comp[i].h + h_max - 1) / h_max;
        z->img_comp[i].y = (s->img_y * z->img_comp[i].v + v_max - 1) / v_max;
        // to simplify generation, we’ll allocate enough memory to decode
        // the bogus oversized data from using interleaved MCUs and their
        // big blocks (e.g. a 16x16 iMCU on an image of width 33); we won’t
        // discard the extra data until colorspace conversion
        //
        // img_mcu_x, img_mcu_y: <=17 bits; comp[i].h and .v are <=4 (checked earlier)
        // so these muls can’t overflow with 32-bit ints (which we require)
        z->img_comp[i].w2 = z->img_mcu_x * z->img_comp[i].h * 8;
        z->img_comp[i].h2 = z->img_mcu_y * z->img_comp[i].v * 8;
        z->img_comp[i].coeff = 0;
        z->img_comp[i].raw_coeff = 0;
        z->img_comp[i].linebuf = NULL;
        z->img_comp[i].raw_data = stbi__malloc_mad2(z->img_comp[i].w2, z->img_comp[i].h2, 15);
        if (z->img_comp[i].raw_data == NULL)
            return stbi__free_jpeg_components(z, i + 1, stbi__err("outofmem", "Out of memory"));
        // align blocks for idct using mmx/sse
        z->img_comp[i].data = (stbi_uc*)(((size_t)z->img_comp[i].raw_data + 15) & ~15);
        if (z->progressive) {
            // w2, h2 are multiples of 8 (see above)
            z->img_comp[i].coeff_w = z->img_comp[i].w2 / 8;
            z->img_comp[i].coeff_h = z->img_comp[i].h2 / 8;
            z->img_comp[i].raw_coeff = stbi__malloc_mad3(z->img_comp[i].w2, z->img_comp[i].h2, sizeof(short), 15);
            if (z->img_comp[i].raw_coeff == NULL)
                return stbi__free_jpeg_components(z, i + 1, stbi__err("outofmem", "Out of memory"));
            z->img_comp[i].coeff = (short*)(((size_t)z->img_comp[i].raw_coeff + 15) & ~15);
        }
    }

    return 1;
}

// use comparisons since in some cases we handle more than one case (e.g. SOF)
#define stbi__DNL(x)         ((x) == 0xdc)
#define stbi__SOI(x)         ((x) == 0xd8)
#define stbi__EOI(x)         ((x) == 0xd9)
#define stbi__SOF(x)         ((x) == 0xc0 || (x) == 0xc1 || (x) == 0xc2)
#define stbi__SOS(x)         ((x) == 0xda)

#define stbi__SOF_progressive(x)   ((x) == 0xc2)

static int stbi__decode_jpeg_header(stbi__jpeg *z, int scan)
{
    int m;
    z->marker = STBI__MARKER_none; // initialize cached marker to empty
    m = stbi__get_marker(z);
    if (!stbi__SOI(m)) return stbi__err("no SOI", "Corrupt JPEG");
    if (scan == STBI__SCAN_type) return 1;
    m = stbi__get_marker(z);
    while (!stbi__SOF(m)) {
        if (!stbi__process_marker(z, m)) return 0;
        m = stbi__get_marker(z);
        while (m == STBI__MARKER_none) {
            // some files have extra padding after their blocks, so ok, we’ll scan
            if (stbi__at_eof(z->s)) return stbi__err("no SOF", "Corrupt JPEG");
            m = stbi__get_marker(z);
        }
    }
    z->progressive = stbi__SOF_progressive(m);
    if (!stbi__process_frame_header(z, scan)) return 0;
    return 1;
}

// decode image to YCbCr format
static int stbi__decode_jpeg_image(stbi__jpeg *j)
{
    int m;
    for (m = 0; m < 4; m++) {
        j->img_comp[m].raw_data = NULL;
        j->img_comp[m].raw_coeff = NULL;
    }
    j->restart_interval = 0;
    if (!stbi__decode_jpeg_header(j, STBI__SCAN_load)) return 0;
    m = stbi__get_marker(j);
    while (!stbi__EOI(m)) {
        if (stbi__SOS(m)) {
            if (!stbi__process_scan_header(j)) return 0;
            if (!stbi__parse_entropy_coded_data(j)) return 0;
            if (j->marker == STBI__MARKER_none) {
                // handle 0s at the end of image data from IP Kamera 9060
                while (!stbi__at_eof(j->s)) {
                    int x = stbi__get8(j->s);
                    if (x == 255) {
                        j->marker = stbi__get8(j->s);
                        break;
                    }
                    else if (x != 0) {
                        return stbi__err("junk before marker", "Corrupt JPEG");
                    }
                }
                // if we reach eof without hitting a marker, stbi__get_marker() below will fail and
   we'll eventually return 0
            }
        }
        else {
            if (!stbi__process_marker(j, m)) return 0;
        }
        m = stbi__get_marker(j);
    }
    if (j->progressive)
        stbi__jpeg_finish(j);
    return 1;
}

// static jfif-centered resampling (across block boundaries)

typedef stbi_uc *(*resample_row_func)(stbi_uc *out, stbi_uc *in0, stbi_uc *in1,
    int w, int hs);

#define stbi__div4(x) ((stbi_uc) ((x) » 2))

static stbi_uc *resample_row_1(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
    STBI_NOTUSED(out);
    STBI_NOTUSED(in_far);
    STBI_NOTUSED(w);
    STBI_NOTUSED(hs);
    return in_near;
}

static stbi_uc* stbi__resample_row_v_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
    // need to generate two samples vertically for every one in input
    int i;
    STBI_NOTUSED(hs);
    for (i = 0; i < w; ++i)
        out[i] = stbi__div4(3 * in_near[i] + in_far[i] + 2);
    return out;
}

static stbi_uc*  stbi__resample_row_h_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
    // need to generate two samples horizontally for every one in input
    int i;
    stbi_uc *input = in_near;

    if (w == 1) {
        // if only one sample, can't do any interpolation
        out[0] = out[1] = input[0];
        return out;
    }

    out[0] = input[0];
    out[1] = stbi__div4(input[0] * 3 + input[1] + 2);
    for (i = 1; i < w - 1; ++i) {
        int n = 3 * input[i] + 2;
        out[i * 2 + 0] = stbi__div4(n + input[i - 1]);
        out[i * 2 + 1] = stbi__div4(n + input[i + 1]);
    }
    out[i * 2 + 0] = stbi__div4(input[w - 2] * 3 + input[w - 1] + 2);
    out[i * 2 + 1] = input[w - 1];

    STBI_NOTUSED(in_far);
    STBI_NOTUSED(hs);

    return out;
}

#define stbi__div16(x) ((stbi_uc) ((x) » 4))

static stbi_uc *stbi__resample_row_hv_2(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
    // need to generate 2x2 samples for every one in input
    int i, t0, t1;
    if (w == 1) {
        out[0] = out[1] = stbi__div4(3 * in_near[0] + in_far[0] + 2);
        return out;
    }

    t1 = 3 * in_near[0] + in_far[0];
    out[0] = stbi__div4(t1 + 2);
    for (i = 1; i < w; ++i) {
        t0 = t1;
        t1 = 3 * in_near[i] + in_far[i];
        out[i * 2 - 1] = stbi__div16(3 * t0 + t1 + 8);
        out[i * 2] = stbi__div16(3 * t1 + t0 + 8);
    }
    out[w * 2 - 1] = stbi__div4(t1 + 2);

    STBI_NOTUSED(hs);

    return out;
}

#if defined(STBI_SSE2) || defined(STBI_NEON)
static stbi_uc *stbi__resample_row_hv_2_simd(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w,
  int hs)
{
    // need to generate 2x2 samples for every one in input
    int i = 0, t0, t1;

    if (w == 1) {
        out[0] = out[1] = stbi__div4(3 * in_near[0] + in_far[0] + 2);
        return out;
    }

    t1 = 3 * in_near[0] + in_far[0];
    // process groups of 8 pixels for as long as we can.
    // note we can't handle the last pixel in a row in this loop
    // because we need to handle the filter boundary conditions.
    for (; i < ((w - 1) & ~7); i += 8) {
#if defined(STBI_SSE2)
        // load and perform the vertical filtering pass
        // this uses 3*x + y = 4*x + (y - x)
        __m128i zero = _mm_setzero_si128();
        __m128i farb = _mm_loadl_epi64((__m128i *) (in_far + i));
        __m128i nearb = _mm_loadl_epi64((__m128i *) (in_near + i));
        __m128i farw = _mm_unpacklo_epi8(farb, zero);
        __m128i nearw = _mm_unpacklo_epi8(nearb, zero);
        __m128i diff = _mm_sub_epi16(farw, nearw);
        __m128i nears = _mm_slli_epi16(nearw, 2);
        __m128i curr = _mm_add_epi16(nears, diff); // current row

                                                   // horizontal filter works the same based on shifted
  vers of current
                                                   // row. "prev" is current row shifted right by 1
  pixel; we need to
                                                   // insert the previous pixel value (from t1).
                                                   // "next" is current row shifted left by 1 pixel,
  with first pixel
                                                   // of next block of 8 pixels added in.
        __m128i prv0 = _mm_slli_si128(curr, 2);
        __m128i nxt0 = _mm_srli_si128(curr, 2);
        __m128i prev = _mm_insert_epi16(prv0, t1, 0);
        __m128i next = _mm_insert_epi16(nxt0, 3 * in_near[i + 8] + in_far[i + 8], 7);

        // horizontal filter, polyphase implementation since it's convenient:
        // even pixels = 3*cur + prev = cur*4 + (prev - cur)
        // odd  pixels = 3*cur + next = cur*4 + (next - cur)
        // note the shared term.
        __m128i bias = _mm_set1_epi16(8);
        __m128i curs = _mm_slli_epi16(curr, 2);
        __m128i prvd = _mm_sub_epi16(prev, curr);
        __m128i nxtd = _mm_sub_epi16(next, curr);
        __m128i curb = _mm_add_epi16(curs, bias);
        __m128i even = _mm_add_epi16(prvd, curb);
        __m128i odd = _mm_add_epi16(nxtd, curb);

        // interleave even and odd pixels, then undo scaling.
        __m128i int0 = _mm_unpacklo_epi16(even, odd);
        __m128i int1 = _mm_unpackhi_epi16(even, odd);
        __m128i de0 = _mm_srli_epi16(int0, 4);
        __m128i de1 = _mm_srli_epi16(int1, 4);

        // pack and write output
        __m128i outv = _mm_packus_epi16(de0, de1);
        _mm_storeu_si128((__m128i *) (out + i * 2), outv);
#elif defined(STBI_NEON)
        // load and perform the vertical filtering pass
        // this uses 3*x + y = 4*x + (y - x)
        uint8x8_t farb = vld1_u8(in_far + i);
        uint8x8_t nearb = vld1_u8(in_near + i);
        int16x8_t diff = vreinterpretq_s16_u16(vsubl_u8(farb, nearb));
        int16x8_t nears = vreinterpretq_s16_u16(vshll_n_u8(nearb, 2));
        int16x8_t curr = vaddq_s16(nears, diff); // current row

                                                 // horizontal filter works the same based on shifted vers of current
                                                 // row. "prev" is current row shifted right by 1 pixel; we need to
                                                 // insert the previous pixel value (from t1).
                                                 // "next" is current row shifted left by 1 pixel, with first pixel
                                                 // of next block of 8 pixels added in.
        int16x8_t prv0 = vextq_s16(curr, curr, 7);
        int16x8_t nxt0 = vextq_s16(curr, curr, 1);
        int16x8_t prev = vsetq_lane_s16(t1, prv0, 0);
        int16x8_t next = vsetq_lane_s16(3 * in_near[i + 8] + in_far[i + 8], nxt0, 7);

        // horizontal filter, polyphase implementation since it's convenient:
        // even pixels = 3*cur + prev = cur*4 + (prev - cur)
        // odd  pixels = 3*cur + next = cur*4 + (next - cur)
        // note the shared term.
        int16x8_t curs = vshlq_n_s16(curr, 2);
        int16x8_t prvd = vsubq_s16(prev, curr);
        int16x8_t nxtd = vsubq_s16(next, curr);
        int16x8_t even = vaddq_s16(curs, prvd);
        int16x8_t odd = vaddq_s16(curs, nxtd);

        // undo scaling and round, then store with even/odd phases interleaved
        uint8x8x2_t o;
        o.val[0] = vqrshrun_n_s16(even, 4);
        o.val[1] = vqrshrun_n_s16(odd, 4);
        vst2_u8(out + i * 2, o);
#endif

        // "previous" value for next iter
        t1 = 3 * in_near[i + 7] + in_far[i + 7];
    }

    t0 = t1;
    t1 = 3 * in_near[i] + in_far[i];
    out[i * 2] = stbi__div16(3 * t1 + t0 + 8);

    for (++i; i < w; ++i) {
        t0 = t1;
        t1 = 3 * in_near[i] + in_far[i];
        out[i * 2 - 1] = stbi__div16(3 * t0 + t1 + 8);
        out[i * 2] = stbi__div16(3 * t1 + t0 + 8);
    }
    out[w * 2 - 1] = stbi__div4(t1 + 2);

    STBI_NOTUSED(hs);

    return out;
}
#endif

static stbi_uc *stbi__resample_row_generic(stbi_uc *out, stbi_uc *in_near, stbi_uc *in_far, int w, int hs)
{
    // resample with nearest-neighbor
    int i, j;
    STBI_NOTUSED(in_far);
    for (i = 0; i < w; ++i)
        for (j = 0; j < hs; ++j)
```

```
            out[i*hs + j] = in_near[i];
    return out;
}

#ifdef STBI_JPEG_OLD
// this is the same YCbCr-to-RGB calculation that stb_image has used
// historically before the algorithm changes in 1.49
#define float2fixed(x)  ((int) ((x) * 65536 + 0.5))
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)
{
    int i;
    for (i = 0; i < count; ++i) {
        int y_fixed = (y[i] << 16) + 32768; // rounding
        int r, g, b;
        int cr = pcr[i] - 128;
        int cb = pcb[i] - 128;
        r = y_fixed + cr*float2fixed(1.40200f);
        g = y_fixed - cr*float2fixed(0.71414f) - cb*float2fixed(0.34414f);
        b = y_fixed + cb*float2fixed(1.77200f);
        r >>= 16;
        g >>= 16;
        b >>= 16;
        if ((unsigned)r > 255) { if (r < 0) r = 0; else r = 255; }
        if ((unsigned)g > 255) { if (g < 0) g = 0; else g = 255; }
        if ((unsigned)b > 255) { if (b < 0) b = 0; else b = 255; }
        out[0] = (stbi_uc)r;
        out[1] = (stbi_uc)g;
        out[2] = (stbi_uc)b;
        out[3] = 255;
        out += step;
    }
}
#else
// this is a reduced-precision calculation of YCbCr-to-RGB introduced
// to make sure the code produces the same results in both SIMD and scalar
#define float2fixed(x)  (((int) ((x) * 4096.0f + 0.5f)) << 8)
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)
{
    int i;
    for (i = 0; i < count; ++i) {
        int y_fixed = (y[i] << 20) + (1 << 19); // rounding
        int r, g, b;
        int cr = pcr[i] - 128;
        int cb = pcb[i] - 128;
        r = y_fixed + cr* float2fixed(1.40200f);
        g = y_fixed + (cr*-float2fixed(0.71414f)) + ((cb*-float2fixed(0.34414f)) & 0xffff0000);
        b = y_fixed + cb* float2fixed(1.77200f);
        r >>= 20;
        g >>= 20;
        b >>= 20;
        if ((unsigned)r > 255) { if (r < 0) r = 0; else r = 255; }
        if ((unsigned)g > 255) { if (g < 0) g = 0; else g = 255; }
        if ((unsigned)b > 255) { if (b < 0) b = 0; else b = 255; }
        out[0] = (stbi_uc)r;
        out[1] = (stbi_uc)g;
        out[2] = (stbi_uc)b;
        out[3] = 255;
        out += step;
    }
}
#endif

#if defined(STBI_SSE2) || defined(STBI_NEON)
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)
{
    int i = 0;

#ifdef STBI_SSE2
    // step == 3 is pretty ugly on the final interleave, and i'm not convinced
    // it's useful in practice (you wouldn't use it for textures, for example).
    // so just accelerate step == 4 case.
    if (step == 4) {
        // this is a fairly straightforward implementation and not super-optimized.
        __m128i signflip = _mm_set1_epi8(-0x80);
        __m128i cr_const0 = _mm_set1_epi16((short)(1.40200f*4096.0f + 0.5f));
        __m128i cr_const1 = _mm_set1_epi16(-(short)(0.71414f*4096.0f + 0.5f));
        __m128i cb_const0 = _mm_set1_epi16(-(short)(0.34414f*4096.0f + 0.5f));
        __m128i cb_const1 = _mm_set1_epi16((short)(1.77200f*4096.0f + 0.5f));
        __m128i y_bias = _mm_set1_epi8((char)(unsigned char)128);
        __m128i xw = _mm_set1_epi16(255); // alpha channel

        for (; i + 7 < count; i += 8) {
            // load
            __m128i y_bytes = _mm_loadl_epi64((__m128i *) (y + i));
            __m128i cr_bytes = _mm_loadl_epi64((__m128i *) (pcr + i));
            __m128i cb_bytes = _mm_loadl_epi64((__m128i *) (pcb + i));
            __m128i cr_biased = _mm_xor_si128(cr_bytes, signflip); // -128
            __m128i cb_biased = _mm_xor_si128(cb_bytes, signflip); // -128

                                                                  // unpack to short (and left-shift
    cr, cb by 8)
            __m128i yw = _mm_unpacklo_epi8(y_bias, y_bytes);
            __m128i crw = _mm_unpacklo_epi8(_mm_setzero_si128(), cr_biased);
            __m128i cbw = _mm_unpacklo_epi8(_mm_setzero_si128(), cb_biased);

            // color transform
            __m128i yws = _mm_srli_epi16(yw, 4);
            __m128i cr0 = _mm_mulhi_epi16(cr_const0, crw);
            __m128i cb0 = _mm_mulhi_epi16(cb_const0, cbw);
            __m128i cb1 = _mm_mulhi_epi16(cbw, cb_const1);
            __m128i cr1 = _mm_mulhi_epi16(crw, cr_const1);
            __m128i rws = _mm_add_epi16(cr0, yws);
            __m128i gwt = _mm_add_epi16(cb0, yws);
            __m128i bws = _mm_add_epi16(yws, cb1);
            __m128i gws = _mm_add_epi16(gwt, cr1);

            // descale
            __m128i rw = _mm_srai_epi16(rws, 4);
            __m128i bw = _mm_srai_epi16(bws, 4);
            __m128i gw = _mm_srai_epi16(gws, 4);

            // back to byte, set up for transpose
            __m128i brb = _mm_packus_epi16(rw, bw);
            __m128i gxb = _mm_packus_epi16(gw, xw);

            // transpose to interleave channels
            __m128i t0 = _mm_unpacklo_epi8(brb, gxb);
            __m128i t1 = _mm_unpackhi_epi8(brb, gxb);
            __m128i o0 = _mm_unpacklo_epi16(t0, t1);
            __m128i o1 = _mm_unpackhi_epi16(t0, t1);

            // store
            _mm_storeu_si128((__m128i *) (out + 0), o0);
            _mm_storeu_si128((__m128i *) (out + 16), o1);
            out += 32;
        }
    }
#endif

#ifdef STBI_NEON
    // in this version, step=3 support would be easy to add. but is there demand?
    if (step == 4) {
        // this is a fairly straightforward implementation and not super-optimized.
        uint8x8_t signflip = vdup_n_u8(0x80);
        int16x8_t cr_const0 = vdupq_n_s16((short)(1.40200f*4096.0f + 0.5f));
        int16x8_t cr_const1 = vdupq_n_s16(-(short)(0.71414f*4096.0f + 0.5f));
        int16x8_t cb_const0 = vdupq_n_s16(-(short)(0.34414f*4096.0f + 0.5f));
        int16x8_t cb_const1 = vdupq_n_s16((short)(1.77200f*4096.0f + 0.5f));

        for (; i + 7 < count; i += 8) {
            // load
            uint8x8_t y_bytes = vld1_u8(y + i);
            uint8x8_t cr_bytes = vld1_u8(pcr + i);
            uint8x8_t cb_bytes = vld1_u8(pcb + i);
            int8x8_t cr_biased = vreinterpret_s8_u8(vsub_u8(cr_bytes, signflip));
            int8x8_t cb_biased = vreinterpret_s8_u8(vsub_u8(cb_bytes, signflip));

            // expand to s16
            int16x8_t yws = vreinterpretq_s16_u16(vshll_n_u8(y_bytes, 4));
            int16x8_t crw = vshll_n_s8(cr_biased, 7);
            int16x8_t cbw = vshll_n_s8(cb_biased, 7);

            // color transform
            int16x8_t cr0 = vqdmulhq_s16(crw, cr_const0);
            int16x8_t cb0 = vqdmulhq_s16(cbw, cb_const0);
            int16x8_t cr1 = vqdmulhq_s16(crw, cr_const1);
            int16x8_t cb1 = vqdmulhq_s16(cbw, cb_const1);
            int16x8_t rws = vaddq_s16(yws, cr0);
            int16x8_t gws = vaddq_s16(vaddq_s16(yws, cb0), cr1);
            int16x8_t bws = vaddq_s16(yws, cb1);

            // undo scaling, round, convert to byte
            uint8x8x4_t o;
            o.val[0] = vqrshrun_n_s16(rws, 4);
            o.val[1] = vqrshrun_n_s16(gws, 4);
            o.val[2] = vqrshrun_n_s16(bws, 4);
            o.val[3] = vdup_n_u8(255);

            // store, interleaving r/g/b/a
            vst4_u8(out, o);
            out += 8 * 4;
        }
    }
#endif

    for (; i < count; ++i) {
        int y_fixed = (y[i] « 20) + (1 « 19); // rounding
        int r, g, b;
        int cr = pcr[i] - 128;
        int cb = pcb[i] - 128;
        r = y_fixed + cr* float2fixed(1.40200f);
        g = y_fixed + cr*-float2fixed(0.71414f) + ((cb*-float2fixed(0.34414f)) & 0xffff0000);
        b = y_fixed + cb* float2fixed(1.77200f);
        r »= 20;
        g »= 20;
        b »= 20;
        if ((unsigned)r > 255) { if (r < 0) r = 0; else r = 255; }
        if ((unsigned)g > 255) { if (g < 0) g = 0; else g = 255; }
        if ((unsigned)b > 255) { if (b < 0) b = 0; else b = 255; }
        out[0] = (stbi_uc)r;
        out[1] = (stbi_uc)g;
        out[2] = (stbi_uc)b;
        out[3] = 255;
        out += step;
    }
}
#endif

// set up the kernels
static void stbi__setup_jpeg(stbi__jpeg *j)
{
    j->idct_block_kernel = stbi__idct_block;
    j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_row;
    j->resample_row_hv_2_kernel = stbi__resample_row_hv_2;

#ifdef STBI_SSE2
    if (stbi__sse2_available()) {
        j->idct_block_kernel = stbi__idct_simd;
#ifndef STBI_JPEG_OLD
        j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_simd;
#endif
        j->resample_row_hv_2_kernel = stbi__resample_row_hv_2_simd;
    }
#endif

#ifdef STBI_NEON
    j->idct_block_kernel = stbi__idct_simd;
#ifndef STBI_JPEG_OLD
    j->YCbCr_to_RGB_kernel = stbi__YCbCr_to_RGB_simd;
#endif
    j->resample_row_hv_2_kernel = stbi__resample_row_hv_2_simd;
#endif
}

// clean up the temporary component buffers
static void stbi__cleanup_jpeg(stbi__jpeg *j)
{
    stbi__free_jpeg_components(j, j->s->img_n, 0);
}

typedef struct
{
    resample_row_func resample;
    stbi_uc *line0, *line1;
    int hs, vs;   // expansion factor in each axis
    int w_lores; // horizontal pixels pre-expansion
    int ystep;   // how far through vertical expansion we are
    int ypos;    // which pre-expansion row we're on
} stbi__resample;

static stbi_uc *load_jpeg_image(stbi__jpeg *z, int *out_x, int *out_y, int *comp, int req_comp)
{
    int n, decode_n;
    z->s->img_n = 0; // make stbi__cleanup_jpeg safe

                     // validate req_comp
    if (req_comp < 0 || req_comp > 4) return stbi__errpuc("bad req_comp", "Internal error");

    // load a jpeg image from whichever source, but leave in YCbCr format
    if (!stbi__decode_jpeg_image(z)) { stbi__cleanup_jpeg(z); return NULL; }

    // determine actual number of components to generate
    n = req_comp ? req_comp : z->s->img_n;

    if (z->s->img_n == 3 && n < 3)
        decode_n = 1;
    else
        decode_n = z->s->img_n;
```

```

   // resample and color-convert
   {
      int k;
      unsigned int i, j;
      stbi_uc *output;
      stbi_uc *coutput[4];

      stbi__resample res_comp[4];

      for (k = 0; k < decode_n; ++k) {
         stbi__resample *r = &res_comp[k];

         // allocate line buffer big enough for upsampling off the edges
         // with upsample factor of 4
         z->img_comp[k].linebuf = (stbi_uc *)stbi__malloc(z->s->img_x + 3);
         if (!z->img_comp[k].linebuf) { stbi__cleanup_jpeg(z); return stbi__errpuc("outofmem", "Out
of memory"); }

         r->hs = z->img_h_max / z->img_comp[k].h;
         r->vs = z->img_v_max / z->img_comp[k].v;
         r->ystep = r->vs » 1;
         r->w_lores = (z->s->img_x + r->hs - 1) / r->hs;
         r->ypos = 0;
         r->line0 = r->line1 = z->img_comp[k].data;

         if (r->hs == 1 && r->vs == 1) r->resample = resample_row_1;
         else if (r->hs == 1 && r->vs == 2) r->resample = stbi__resample_row_v_2;
         else if (r->hs == 2 && r->vs == 1) r->resample = stbi__resample_row_h_2;
         else if (r->hs == 2 && r->vs == 2) r->resample = z->resample_row_hv_2_kernel;
         else                               r->resample = stbi__resample_row_generic;
      }

      // can't error after this so, this is safe
      output = (stbi_uc *)stbi__malloc_mad3(n, z->s->img_x, z->s->img_y, 1);
      if (!output) { stbi__cleanup_jpeg(z); return stbi__errpuc("outofmem", "Out of memory"); }

      // now go ahead and resample
      for (j = 0; j < z->s->img_y; ++j) {
         stbi_uc *out = output + n * z->s->img_x * j;
         for (k = 0; k < decode_n; ++k) {
            stbi__resample *r = &res_comp[k];
            int y_bot = r->ystep >= (r->vs » 1);
            coutput[k] = r->resample(z->img_comp[k].linebuf,
               y_bot ? r->line1 : r->line0,
               y_bot ? r->line0 : r->line1,
               r->w_lores, r->hs);
            if (++r->ystep >= r->vs) {
               r->ystep = 0;
               r->line0 = r->line1;
               if (++r->ypos < z->img_comp[k].y)
                  r->line1 += z->img_comp[k].w2;
            }
         }
         if (n >= 3) {
            stbi_uc *y = coutput[0];
            if (z->s->img_n == 3) {
               if (z->rgb == 3) {
                  for (i = 0; i < z->s->img_x; ++i) {
                     out[0] = y[i];
                     out[1] = coutput[1][i];
                     out[2] = coutput[2][i];
                     out[3] = 255;
                     out += n;
                  }
               }
               else {
                  z->YCbCr_to_RGB_kernel(out, y, coutput[1], coutput[2], z->s->img_x, n);
               }
            }
            else
               for (i = 0; i < z->s->img_x; ++i) {
                  out[0] = out[1] = out[2] = y[i];
                  out[3] = 255; // not used if n==3
                  out += n;
               }
         }
         else {
            stbi_uc *y = coutput[0];
            if (n == 1)
               for (i = 0; i < z->s->img_x; ++i) out[i] = y[i];
            else
               for (i = 0; i < z->s->img_x; ++i) *out++ = y[i], *out++ = 255;
         }
      }
      stbi__cleanup_jpeg(z);
      *out_x = z->s->img_x;
```

```
        *out_y = z->s->img_y;
        if (comp) *comp = z->s->img_n; // report original components, not output
        return output;
    }
}

static void *stbi__jpeg_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri)
{
    unsigned char* result;
    stbi__jpeg* j = (stbi__jpeg*)stbi__malloc(sizeof(stbi__jpeg));
    j->s = s;
    stbi__setup_jpeg(j);
    result = load_jpeg_image(j, x, y, comp, req_comp);
    STBI_FREE(j);
    return result;
}

static int stbi__jpeg_test(stbi__context *s)
{
    int r;
    stbi__jpeg j;
    j.s = s;
    stbi__setup_jpeg(&j);
    r = stbi__decode_jpeg_header(&j, STBI__SCAN_type);
    stbi__rewind(s);
    return r;
}

static int stbi__jpeg_info_raw(stbi__jpeg *j, int *x, int *y, int *comp)
{
    if (!stbi__decode_jpeg_header(j, STBI__SCAN_header)) {
        stbi__rewind(j->s);
        return 0;
    }
    if (x) *x = j->s->img_x;
    if (y) *y = j->s->img_y;
    if (comp) *comp = j->s->img_n;
    return 1;
}

static int stbi__jpeg_info(stbi__context *s, int *x, int *y, int *comp)
{
    int result;
    stbi__jpeg* j = (stbi__jpeg*)(stbi__malloc(sizeof(stbi__jpeg)));
    j->s = s;
    result = stbi__jpeg_info_raw(j, x, y, comp);
    STBI_FREE(j);
    return result;
}
#endif

// public domain zlib decode    v0.2  Sean Barrett 2006-11-18
//    simple implementation
//      - all input must be provided in an upfront buffer
//      - all output is written to a single output buffer (can malloc/realloc)
//    performance
//      - fast huffman

#ifndef STBI_NO_ZLIB

// fast-way is faster to check than jpeg huffman, but slow way is slower
#define STBI__ZFAST_BITS  9 // accelerate all cases in default tables
#define STBI__ZFAST_MASK  ((1 « STBI__ZFAST_BITS) - 1)

// zlib-style huffman encoding
// (jpegs packs from left, zlib from right, so can't share code)
typedef struct
{
    stbi__uint16 fast[1 « STBI__ZFAST_BITS];
    stbi__uint16 firstcode[16];
    int maxcode[17];
    stbi__uint16 firstsymbol[16];
    stbi_uc  size[288];
    stbi__uint16 value[288];
} stbi__zhuffman;

stbi_inline static int stbi__bitreverse16(int n)
{
    n = ((n & 0xAAAA) » 1) | ((n & 0x5555) « 1);
    n = ((n & 0xCCCC) » 2) | ((n & 0x3333) « 2);
    n = ((n & 0xF0F0) » 4) | ((n & 0x0F0F) « 4);
    n = ((n & 0xFF00) » 8) | ((n & 0x00FF) « 8);
    return n;
}

stbi_inline static int stbi__bit_reverse(int v, int bits)
```

```
{
    STBI_ASSERT(bits <= 16);
    // to bit reverse n bits, reverse 16 and shift
    // e.g. 11 bits, bit reverse and shift away 5
    return stbi__bitreverse16(v) » (16 - bits);
}

static int stbi__zbuild_huffman(stbi__zhuffman *z, stbi_uc *sizelist, int num)
{
    int i, k = 0;
    int code, next_code[16], sizes[17];

    // DEFLATE spec for generating codes
    memset(sizes, 0, sizeof(sizes));
    memset(z->fast, 0, sizeof(z->fast));
    for (i = 0; i < num; ++i)
        ++sizes[sizelist[i]];
    sizes[0] = 0;
    for (i = 1; i < 16; ++i)
        if (sizes[i] >(1 « i))
            return stbi__err("bad sizes", "Corrupt PNG");
    code = 0;
    for (i = 1; i < 16; ++i) {
        next_code[i] = code;
        z->firstcode[i] = (stbi__uint16)code;
        z->firstsymbol[i] = (stbi__uint16)k;
        code = (code + sizes[i]);
        if (sizes[i])
            if (code - 1 >= (1 « i)) return stbi__err("bad codelengths", "Corrupt PNG");
        z->maxcode[i] = code « (16 - i); // preshift for inner loop
        code «= 1;
        k += sizes[i];
    }
    z->maxcode[16] = 0x10000; // sentinel
    for (i = 0; i < num; ++i) {
        int s = sizelist[i];
        if (s) {
            int c = next_code[s] - z->firstcode[s] + z->firstsymbol[s];
            stbi__uint16 fastv = (stbi__uint16)((s « 9) | i);
            z->size[c] = (stbi_uc)s;
            z->value[c] = (stbi__uint16)i;
            if (s <= STBI__ZFAST_BITS) {
                int j = stbi__bit_reverse(next_code[s], s);
                while (j < (1 « STBI__ZFAST_BITS)) {
                    z->fast[j] = fastv;
                    j += (1 « s);
                }
            }
            ++next_code[s];
        }
    }
    return 1;
}

// zlib-from-memory implementation for PNG reading
//    because PNG allows splitting the zlib stream arbitrarily,
//    and it's annoying structurally to have PNG call ZLIB call PNG,
//    we require PNG read all the IDATs and combine them into a single
//    memory buffer

typedef struct
{
    stbi_uc *zbuffer, *zbuffer_end;
    int num_bits;
    stbi__uint32 code_buffer;

    char *zout;
    char *zout_start;
    char *zout_end;
    int   z_expandable;

    stbi__zhuffman z_length, z_distance;
} stbi__zbuf;

stbi_inline static stbi_uc stbi__zget8(stbi__zbuf *z)
{
    if (z->zbuffer >= z->zbuffer_end) return 0;
    return *z->zbuffer++;
}

static void stbi__fill_bits(stbi__zbuf *z)
{
    do {
        STBI_ASSERT(z->code_buffer < (1U « z->num_bits));
        z->code_buffer |= (unsigned int)stbi__zget8(z) « z->num_bits;
        z->num_bits += 8;
    } while (z->num_bits <= 24);
```

```
}

stbi_inline static unsigned int stbi__zreceive(stbi__zbuf *z, int n)
{
    unsigned int k;
    if (z->num_bits < n) stbi__fill_bits(z);
    k = z->code_buffer & ((1 « n) - 1);
    z->code_buffer »= n;
    z->num_bits -= n;
    return k;
}

static int stbi__zhuffman_decode_slowpath(stbi__zbuf *a, stbi__zhuffman *z)
{
    int b, s, k;
    // not resolved by fast table, so compute it the slow way
    // use jpeg approach, which requires MSbits at top
    k = stbi__bit_reverse(a->code_buffer, 16);
    for (s = STBI__ZFAST_BITS + 1; ; ++s)
        if (k < z->maxcode[s])
            break;
    if (s == 16) return -1; // invalid code!
                            // code size is s, so:
    b = (k » (16 - s)) - z->firstcode[s] + z->firstsymbol[s];
    STBI_ASSERT(z->size[b] == s);
    a->code_buffer »= s;
    a->num_bits -= s;
    return z->value[b];
}

stbi_inline static int stbi__zhuffman_decode(stbi__zbuf *a, stbi__zhuffman *z)
{
    int b, s;
    if (a->num_bits < 16) stbi__fill_bits(a);
    b = z->fast[a->code_buffer & STBI__ZFAST_MASK];
    if (b) {
        s = b » 9;
        a->code_buffer »= s;
        a->num_bits -= s;
        return b & 511;
    }
    return stbi__zhuffman_decode_slowpath(a, z);
}

static int stbi__zexpand(stbi__zbuf *z, char *zout, int n)  // need to make room for n bytes
{
    char *q;
    int cur, limit, old_limit;
    z->zout = zout;
    if (!z->z_expandable) return stbi__err("output buffer limit", "Corrupt PNG");
    cur = (int)(z->zout - z->zout_start);
    limit = old_limit = (int)(z->zout_end - z->zout_start);
    while (cur + n > limit)
        limit *= 2;
    q = (char *)STBI_REALLOC_SIZED(z->zout_start, old_limit, limit);
    STBI_NOTUSED(old_limit);
    if (q == NULL) return stbi__err("outofmem", "Out of memory");
    z->zout_start = q;
    z->zout = q + cur;
    z->zout_end = q + limit;
    return 1;
}

static int stbi__zlength_base[31] = {
    3,4,5,6,7,8,9,10,11,13,
    15,17,19,23,27,31,35,43,51,59,
    67,83,99,115,131,163,195,227,258,0,0 };

static int stbi__zlength_extra[31] =
{ 0,0,0,0,0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,0,0,0 };

static int stbi__zdist_base[32] = { 1,2,3,4,5,7,9,13,17,25,33,49,65,97,129,193,
257,385,513,769,1025,1537,2049,3073,4097,6145,8193,12289,16385,24577,0,0 };

static int stbi__zdist_extra[32] =
{ 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 };

static int stbi__parse_huffman_block(stbi__zbuf *a)
{
    char *zout = a->zout;
    for (;;) {
        int z = stbi__zhuffman_decode(a, &a->z_length);
        if (z < 256) {
            if (z < 0) return stbi__err("bad huffman code", "Corrupt PNG"); // error in huffman codes
            if (zout >= a->zout_end) {
                if (!stbi__zexpand(a, zout, 1)) return 0;
                zout = a->zout;
            }
            *zout++ = (char)z;
        }
        else {
            stbi_uc *p;
            int len, dist;
            if (z == 256) {
                a->zout = zout;
                return 1;
            }
            z -= 257;
            len = stbi__zlength_base[z];
            if (stbi__zlength_extra[z]) len += stbi__zreceive(a, stbi__zlength_extra[z]);
            z = stbi__zhuffman_decode(a, &a->z_distance);
            if (z < 0) return stbi__err("bad huffman code", "Corrupt PNG");
            dist = stbi__zdist_base[z];
            if (stbi__zdist_extra[z]) dist += stbi__zreceive(a, stbi__zdist_extra[z]);
            if (zout - a->zout_start < dist) return stbi__err("bad dist", "Corrupt PNG");
            if (zout + len > a->zout_end) {
                if (!stbi__zexpand(a, zout, len)) return 0;
                zout = a->zout;
            }
            p = (stbi_uc *)(zout - dist);
            if (dist == 1) { // run of one byte; common in images.
                stbi_uc v = *p;
                if (len) { do *zout++ = v; while (--len); }
            }
            else {
                if (len) { do *zout++ = *p++; while (--len); }
            }
        }
    }
}

static int stbi__compute_huffman_codes(stbi__zbuf *a)
{
    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 };
    stbi__zhuffman z_codelength;
    stbi_uc lencodes[286 + 32 + 137];//padding for maximum single op
    stbi_uc codelength_sizes[19];
    int i, n;

    int hlit = stbi__zreceive(a, 5) + 257;
    int hdist = stbi__zreceive(a, 5) + 1;
    int hclen = stbi__zreceive(a, 4) + 4;
    int ntot = hlit + hdist;

    memset(codelength_sizes, 0, sizeof(codelength_sizes));
    for (i = 0; i < hclen; ++i) {
        int s = stbi__zreceive(a, 3);
        codelength_sizes[length_dezigzag[i]] = (stbi_uc)s;
    }
    if (!stbi__zbuild_huffman(&z_codelength, codelength_sizes, 19)) return 0;

    n = 0;
    while (n < ntot) {
        int c = stbi__zhuffman_decode(a, &z_codelength);
        if (c < 0 || c >= 19) return stbi__err("bad codelengths", "Corrupt PNG");
        if (c < 16)
            lencodes[n++] = (stbi_uc)c;
        else {
            stbi_uc fill = 0;
            if (c == 16) {
                c = stbi__zreceive(a, 2) + 3;
                if (n == 0) return stbi__err("bad codelengths", "Corrupt PNG");
                fill = lencodes[n - 1];
            }
            else if (c == 17)
                c = stbi__zreceive(a, 3) + 3;
            else {
                STBI_ASSERT(c == 18);
                c = stbi__zreceive(a, 7) + 11;
            }
            if (ntot - n < c) return stbi__err("bad codelengths", "Corrupt PNG");
            memset(lencodes + n, fill, c);
            n += c;
        }
    }
    if (n != ntot) return stbi__err("bad codelengths", "Corrupt PNG");
    if (!stbi__zbuild_huffman(&a->z_length, lencodes, hlit)) return 0;
    if (!stbi__zbuild_huffman(&a->z_distance, lencodes + hlit, hdist)) return 0;
    return 1;
}

static int stbi__parse_uncompressed_block(stbi__zbuf *a)
{
    stbi_uc header[4];
    int len, nlen, k;
    if (a->num_bits & 7)
        stbi__zreceive(a, a->num_bits & 7); // discard
                                            // drain the bit-packed data into header
    k = 0;
    while (a->num_bits > 0) {
        header[k++] = (stbi_uc)(a->code_buffer & 255); // suppress MSVC run-time check
        a->code_buffer >>= 8;
        a->num_bits -= 8;
    }
    STBI_ASSERT(a->num_bits == 0);
    // now fill header the normal way
    while (k < 4)
        header[k++] = stbi__zget8(a);
    len = header[1] * 256 + header[0];
    nlen = header[3] * 256 + header[2];
    if (nlen != (len ^ 0xffff)) return stbi__err("zlib corrupt", "Corrupt PNG");
    if (a->zbuffer + len > a->zbuffer_end) return stbi__err("read past buffer", "Corrupt PNG");
    if (a->zout + len > a->zout_end)
        if (!stbi__zexpand(a, a->zout, len)) return 0;
    memcpy(a->zout, a->zbuffer, len);
    a->zbuffer += len;
    a->zout += len;
    return 1;
}

static int stbi__parse_zlib_header(stbi__zbuf *a)
{
    int cmf = stbi__zget8(a);
    int cm = cmf & 15;
    /* int cinfo = cmf >> 4; */
    int flg = stbi__zget8(a);
    if ((cmf * 256 + flg) % 31 != 0) return stbi__err("bad zlib header", "Corrupt PNG"); // zlib spec
    if (flg & 32) return stbi__err("no preset dict", "Corrupt PNG"); // preset dictionary not allowed
  in png
    if (cm != 8) return stbi__err("bad compression", "Corrupt PNG"); // DEFLATE required for png
                                                                    // window = 1 << (8 + cinfo)... but
  who cares, we fully buffer output
    return 1;
}

// @TODO: should statically initialize these for optimal thread safety
static stbi_uc stbi__zdefault_length[288], stbi__zdefault_distance[32];
static void stbi__init_zdefaults(void)
{
    int i;   // use <= to match clearly with spec
    for (i = 0; i <= 143; ++i)     stbi__zdefault_length[i] = 8;
    for (; i <= 255; ++i)     stbi__zdefault_length[i] = 9;
    for (; i <= 279; ++i)     stbi__zdefault_length[i] = 7;
    for (; i <= 287; ++i)     stbi__zdefault_length[i] = 8;

    for (i = 0; i <= 31; ++i)     stbi__zdefault_distance[i] = 5;
}

static int stbi__parse_zlib(stbi__zbuf *a, int parse_header)
{
    int final, type;
    if (parse_header)
        if (!stbi__parse_zlib_header(a)) return 0;
    a->num_bits = 0;
    a->code_buffer = 0;
    do {
        final = stbi__zreceive(a, 1);
        type = stbi__zreceive(a, 2);
        if (type == 0) {
            if (!stbi__parse_uncompressed_block(a)) return 0;
        }
        else if (type == 3) {
            return 0;
        }
        else {
            if (type == 1) {
                // use fixed code lengths
                if (!stbi__zdefault_distance[31]) stbi__init_zdefaults();
                if (!stbi__zbuild_huffman(&a->z_length, stbi__zdefault_length, 288)) return 0;
                if (!stbi__zbuild_huffman(&a->z_distance, stbi__zdefault_distance, 32)) return 0;
            }
            else {
                if (!stbi__compute_huffman_codes(a)) return 0;
            }
            if (!stbi__parse_huffman_block(a)) return 0;
        }
    } while (!final);
    return 1;
}

static int stbi__do_zlib(stbi__zbuf *a, char *obuf, int olen, int exp, int parse_header)
```

```
{
    a->zout_start = obuf;
    a->zout = obuf;
    a->zout_end = obuf + olen;
    a->z_expandable = exp;

    return stbi__parse_zlib(a, parse_header);
}

STBIDEF char *stbi_zlib_decode_malloc_guesssize(const char *buffer, int len, int initial_size, int
   *outlen)
{
    stbi__zbuf a;
    char *p = (char *)stbi__malloc(initial_size);
    if (p == NULL) return NULL;
    a.zbuffer = (stbi_uc *)buffer;
    a.zbuffer_end = (stbi_uc *)buffer + len;
    if (stbi__do_zlib(&a, p, initial_size, 1, 1)) {
        if (outlen) *outlen = (int)(a.zout - a.zout_start);
        return a.zout_start;
    }
    else {
        STBI_FREE(a.zout_start);
        return NULL;
    }
}

STBIDEF char *stbi_zlib_decode_malloc(char const *buffer, int len, int *outlen)
{
    return stbi_zlib_decode_malloc_guesssize(buffer, len, 16384, outlen);
}

STBIDEF char *stbi_zlib_decode_malloc_guesssize_headerflag(const char *buffer, int len, int
   initial_size, int *outlen, int parse_header)
{
    stbi__zbuf a;
    char *p = (char *)stbi__malloc(initial_size);
    if (p == NULL) return NULL;
    a.zbuffer = (stbi_uc *)buffer;
    a.zbuffer_end = (stbi_uc *)buffer + len;
    if (stbi__do_zlib(&a, p, initial_size, 1, parse_header)) {
        if (outlen) *outlen = (int)(a.zout - a.zout_start);
        return a.zout_start;
    }
    else {
        STBI_FREE(a.zout_start);
        return NULL;
    }
}

STBIDEF int stbi_zlib_decode_buffer(char *obuffer, int olen, char const *ibuffer, int ilen)
{
    stbi__zbuf a;
    a.zbuffer = (stbi_uc *)ibuffer;
    a.zbuffer_end = (stbi_uc *)ibuffer + ilen;
    if (stbi__do_zlib(&a, obuffer, olen, 0, 1))
        return (int)(a.zout - a.zout_start);
    else
        return -1;
}

STBIDEF char *stbi_zlib_decode_noheader_malloc(char const *buffer, int len, int *outlen)
{
    stbi__zbuf a;
    char *p = (char *)stbi__malloc(16384);
    if (p == NULL) return NULL;
    a.zbuffer = (stbi_uc *)buffer;
    a.zbuffer_end = (stbi_uc *)buffer + len;
    if (stbi__do_zlib(&a, p, 16384, 1, 0)) {
        if (outlen) *outlen = (int)(a.zout - a.zout_start);
        return a.zout_start;
    }
    else {
        STBI_FREE(a.zout_start);
        return NULL;
    }
}

STBIDEF int stbi_zlib_decode_noheader_buffer(char *obuffer, int olen, const char *ibuffer, int ilen)
{
    stbi__zbuf a;
    a.zbuffer = (stbi_uc *)ibuffer;
    a.zbuffer_end = (stbi_uc *)ibuffer + ilen;
    if (stbi__do_zlib(&a, obuffer, olen, 0, 0))
        return (int)(a.zout - a.zout_start);
    else
        return -1;
```

```
}
#endif

// public domain "baseline" PNG decoder   v0.10  Sean Barrett 2006-11-18
//    simple implementation
//      - only 8-bit samples
//      - no CRC checking
//      - allocates lots of intermediate memory
//        - avoids problem of streaming data between subsystems
//        - avoids explicit window management
//    performance
//      - uses stb_zlib, a PD zlib implementation with fast huffman decoding

#ifndef STBI_NO_PNG
typedef struct
{
    stbi__uint32 length;
    stbi__uint32 type;
} stbi__pngchunk;

static stbi__pngchunk stbi__get_chunk_header(stbi__context *s)
{
    stbi__pngchunk c;
    c.length = stbi__get32be(s);
    c.type = stbi__get32be(s);
    return c;
}

static int stbi__check_png_header(stbi__context *s)
{
    static stbi_uc png_sig[8] = { 137,80,78,71,13,10,26,10 };
    int i;
    for (i = 0; i < 8; ++i)
        if (stbi__get8(s) != png_sig[i]) return stbi__err("bad png sig", "Not a PNG");
    return 1;
}

typedef struct
{
    stbi__context *s;
    stbi_uc *idata, *expanded, *out;
    int depth;
} stbi__png;


enum {
    STBI__F_none = 0,
    STBI__F_sub = 1,
    STBI__F_up = 2,
    STBI__F_avg = 3,
    STBI__F_paeth = 4,
    // synthetic filters used for first scanline to avoid needing a dummy row of 0s
    STBI__F_avg_first,
    STBI__F_paeth_first
};

static stbi_uc first_row_filter[5] =
{
    STBI__F_none,
    STBI__F_sub,
    STBI__F_none,
    STBI__F_avg_first,
    STBI__F_paeth_first
};

static int stbi__paeth(int a, int b, int c)
{
    int p = a + b - c;
    int pa = abs(p - a);
    int pb = abs(p - b);
    int pc = abs(p - c);
    if (pa <= pb && pa <= pc) return a;
    if (pb <= pc) return b;
    return c;
}

static stbi_uc stbi__depth_scale_table[9] = { 0, 0xff, 0x55, 0, 0x11, 0,0,0, 0x01 };

// create the png data from post-deflated data
static int stbi__create_png_image_raw(stbi__png *a, stbi_uc *raw, stbi__uint32 raw_len, int out_n,
  stbi__uint32 x, stbi__uint32 y, int depth, int color)
{
    int bytes = (depth == 16 ? 2 : 1);
    stbi__context *s = a->s;
    stbi__uint32 i, j, stride = x*out_n*bytes;
    stbi__uint32 img_len, img_width_bytes;
    int k;
```

```
   int img_n = s->img_n; // copy it into a local for later

   int output_bytes = out_n*bytes;
   int filter_bytes = img_n*bytes;
   int width = x;

   STBI_ASSERT(out_n == s->img_n || out_n == s->img_n + 1);
   a->out = (stbi_uc *)stbi__malloc_mad3(x, y, output_bytes, 0); // extra bytes to write off the end
  into
   if (!a->out) return stbi__err("outofmem", "Out of memory");

   img_width_bytes = (((img_n * x * depth) + 7) » 3);
   img_len = (img_width_bytes + 1) * y;
   if (s->img_x == x && s->img_y == y) {
       if (raw_len != img_len) return stbi__err("not enough pixels", "Corrupt PNG");
   }
   else { // interlaced:
       if (raw_len < img_len) return stbi__err("not enough pixels", "Corrupt PNG");
   }

   for (j = 0; j < y; ++j) {
       stbi_uc *cur = a->out + stride*j;
       stbi_uc *prior = cur - stride;
       int filter = *raw++;

       if (filter > 4)
           return stbi__err("invalid filter", "Corrupt PNG");

       if (depth < 8) {
           STBI_ASSERT(img_width_bytes <= x);
           cur += x*out_n - img_width_bytes; // store output to the rightmost img_len bytes, so we can
  decode in place
           filter_bytes = 1;
           width = img_width_bytes;
       }

       // if first row, use special filter that doesn't sample previous row
       if (j == 0) filter = first_row_filter[filter];

       // handle first byte explicitly
       for (k = 0; k < filter_bytes; ++k) {
           switch (filter) {
           case STBI__F_none: cur[k] = raw[k]; break;
           case STBI__F_sub: cur[k] = raw[k]; break;
           case STBI__F_up: cur[k] = STBI__BYTECAST(raw[k] + prior[k]); break;
           case STBI__F_avg: cur[k] = STBI__BYTECAST(raw[k] + (prior[k] » 1)); break;
           case STBI__F_paeth: cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(0, prior[k], 0)); break;
           case STBI__F_avg_first: cur[k] = raw[k]; break;
           case STBI__F_paeth_first: cur[k] = raw[k]; break;
           }
       }

       if (depth == 8) {
           if (img_n != out_n)
               cur[img_n] = 255; // first pixel
           raw += img_n;
           cur += out_n;
           prior += out_n;
       }
       else if (depth == 16) {
           if (img_n != out_n) {
               cur[filter_bytes] = 255; // first pixel top byte
               cur[filter_bytes + 1] = 255; // first pixel bottom byte
           }
           raw += filter_bytes;
           cur += output_bytes;
           prior += output_bytes;
       }
       else {
           raw += 1;
           cur += 1;
           prior += 1;
       }

       // this is a little gross, so that we don't switch per-pixel or per-component
       if (depth < 8 || img_n == out_n) {
           int nk = (width - 1)*filter_bytes;
#define STBI__CASE(f) \
            case f:     \
               for (k=0; k < nk; ++k)
           switch (filter) {
               // "none" filter turns into a memcpy here; make that explicit.
           case STBI__F_none:         memcpy(cur, raw, nk); break;
               STBI__CASE(STBI__F_sub) { cur[k] = STBI__BYTECAST(raw[k] + cur[k - filter_bytes]); }
  break;
               STBI__CASE(STBI__F_up) { cur[k] = STBI__BYTECAST(raw[k] + prior[k]); } break;
               STBI__CASE(STBI__F_avg) { cur[k] = STBI__BYTECAST(raw[k] + ((prior[k] + cur[k -
filter_bytes]) » 1)); } break;
                STBI__CASE(STBI__F_paeth) { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k - filter_bytes], prior[k], prior[k - filter_bytes])); } break;
                STBI__CASE(STBI__F_avg_first) { cur[k] = STBI__BYTECAST(raw[k] + (cur[k - filter_bytes] » 1)); } break;
                STBI__CASE(STBI__F_paeth_first) { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k - filter_bytes], 0, 0)); } break;
            }
#undef STBI__CASE
            raw += nk;
        }
        else {
            STBI_ASSERT(img_n + 1 == out_n);
#define STBI__CASE(f) \
             case f:     \
                for (i=x-1; i >= 1; --i, cur[filter_bytes]=255,raw+=filter_bytes,cur+=output_bytes,prior+=output_bytes) \
                   for (k=0; k < filter_bytes; ++k)
            switch (filter) {
                STBI__CASE(STBI__F_none) { cur[k] = raw[k]; } break;
                STBI__CASE(STBI__F_sub) { cur[k] = STBI__BYTECAST(raw[k] + cur[k - output_bytes]); } break;
                STBI__CASE(STBI__F_up) { cur[k] = STBI__BYTECAST(raw[k] + prior[k]); } break;
                STBI__CASE(STBI__F_avg) { cur[k] = STBI__BYTECAST(raw[k] + ((prior[k] + cur[k - output_bytes]) » 1)); } break;
                STBI__CASE(STBI__F_paeth) { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k - output_bytes], prior[k], prior[k - output_bytes])); } break;
                STBI__CASE(STBI__F_avg_first) { cur[k] = STBI__BYTECAST(raw[k] + (cur[k - output_bytes] » 1)); } break;
                STBI__CASE(STBI__F_paeth_first) { cur[k] = STBI__BYTECAST(raw[k] + stbi__paeth(cur[k - output_bytes], 0, 0)); } break;
            }
#undef STBI__CASE

            // the loop above sets the high byte of the pixels' alpha, but for
            // 16 bit png files we also need the low byte set. we'll do that here.
            if (depth == 16) {
                cur = a->out + stride*j; // start at the beginning of the row again
                for (i = 0; i < x; ++i, cur += output_bytes) {
                    cur[filter_bytes + 1] = 255;
                }
            }
        }
    }

    // we make a separate pass to expand bits to pixels; for performance,
    // this could run two scanlines behind the above code, so it won't
    // intefere with filtering but will still be in the cache.
    if (depth < 8) {
        for (j = 0; j < y; ++j) {
            stbi_uc *cur = a->out + stride*j;
            stbi_uc *in = a->out + stride*j + x*out_n - img_width_bytes;
            // 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
            // 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
            stbi_uc scale = (color == 0) ? stbi__depth_scale_table[depth] : 1; // scale grayscale values to 0..255 range

                                                                               // note that the final byte might overshoot and write more data than desired.
                                                                               // we can allocate enough data that this never writes out of memory, but it
                                                                               // could also overwrite the next scanline. can it overwrite non-empty data
                                                                               // on the next scanline? yes, consider 1-pixel-wide scanlines with 1-bit-per-pixel.
                                                                               // so we need to explicitly clamp the final ones

            if (depth == 4) {
                for (k = x*img_n; k >= 2; k -= 2, ++in) {
                    *cur++ = scale * ((*in » 4));
                    *cur++ = scale * ((*in) & 0x0f);
                }
                if (k > 0) *cur++ = scale * ((*in » 4));
            }
            else if (depth == 2) {
                for (k = x*img_n; k >= 4; k -= 4, ++in) {
                    *cur++ = scale * ((*in » 6));
                    *cur++ = scale * ((*in » 4) & 0x03);
                    *cur++ = scale * ((*in » 2) & 0x03);
                    *cur++ = scale * ((*in) & 0x03);
                }
                if (k > 0) *cur++ = scale * ((*in » 6));
                if (k > 1) *cur++ = scale * ((*in » 4) & 0x03);
                if (k > 2) *cur++ = scale * ((*in » 2) & 0x03);
            }
            else if (depth == 1) {
                for (k = x*img_n; k >= 8; k -= 8, ++in) {
                    *cur++ = scale * ((*in » 7));
                    *cur++ = scale * ((*in » 6) & 0x01);
                    *cur++ = scale * ((*in » 5) & 0x01);
                    *cur++ = scale * ((*in » 4) & 0x01);
                    *cur++ = scale * ((*in » 3) & 0x01);
                    *cur++ = scale * ((*in » 2) & 0x01);
                    *cur++ = scale * ((*in » 1) & 0x01);
                    *cur++ = scale * ((*in) & 0x01);
                }
                if (k > 0) *cur++ = scale * ((*in » 7));
                if (k > 1) *cur++ = scale * ((*in » 6) & 0x01);
                if (k > 2) *cur++ = scale * ((*in » 5) & 0x01);
                if (k > 3) *cur++ = scale * ((*in » 4) & 0x01);
                if (k > 4) *cur++ = scale * ((*in » 3) & 0x01);
                if (k > 5) *cur++ = scale * ((*in » 2) & 0x01);
                if (k > 6) *cur++ = scale * ((*in » 1) & 0x01);
            }
            if (img_n != out_n) {
                int q;
                // insert alpha = 255
                cur = a->out + stride*j;
                if (img_n == 1) {
                    for (q = x - 1; q >= 0; --q) {
                        cur[q * 2 + 1] = 255;
                        cur[q * 2 + 0] = cur[q];
                    }
                }
                else {
                    STBI_ASSERT(img_n == 3);
                    for (q = x - 1; q >= 0; --q) {
                        cur[q * 4 + 3] = 255;
                        cur[q * 4 + 2] = cur[q * 3 + 2];
                        cur[q * 4 + 1] = cur[q * 3 + 1];
                        cur[q * 4 + 0] = cur[q * 3 + 0];
                    }
                }
            }
        }
    }
    else if (depth == 16) {
        // force the image data from big-endian to platform-native.
        // this is done in a separate pass due to the decoding relying
        // on the data being untouched, but could probably be done
        // per-line during decode if care is taken.
        stbi_uc *cur = a->out;
        stbi__uint16 *cur16 = (stbi__uint16*)cur;

        for (i = 0; i < x*y*out_n; ++i, cur16++, cur += 2) {
            *cur16 = (cur[0] « 8) | cur[1];
        }
    }

    return 1;
}

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)
{
    int bytes = (depth == 16 ? 2 : 1);
    int out_bytes = out_n * bytes;
    stbi_uc *final;
    int p;
    if (!interlaced)
        return stbi__create_png_image_raw(a, image_data, image_data_len, out_n, a->s->img_x,
  a->s->img_y, depth, color);

    // de-interlacing
    final = (stbi_uc *)stbi__malloc_mad3(a->s->img_x, a->s->img_y, out_bytes, 0);
    for (p = 0; p < 7; ++p) {
        int xorig[] = { 0,4,0,2,0,1,0 };
        int yorig[] = { 0,0,4,0,2,0,1 };
        int xspc[] = { 8,8,4,4,2,2,1 };
        int yspc[] = { 8,8,8,4,4,2,2 };
        int i, j, x, y;
        // pass1_x[4] = 0, pass1_x[5] = 1, pass1_x[12] = 1
        x = (a->s->img_x - xorig[p] + xspc[p] - 1) / xspc[p];
        y = (a->s->img_y - yorig[p] + yspc[p] - 1) / yspc[p];
        if (x && y) {
            stbi__uint32 img_len = ((((a->s->img_n * x * depth) + 7) » 3) + 1) * y;
            if (!stbi__create_png_image_raw(a, image_data, image_data_len, out_n, x, y, depth, color))
  {
                STBI_FREE(final);
                return 0;
            }
            for (j = 0; j < y; ++j) {
                for (i = 0; i < x; ++i) {
                    int out_y = j*yspc[p] + yorig[p];
                    int out_x = i*xspc[p] + xorig[p];
                    memcpy(final + out_y*a->s->img_x*out_bytes + out_x*out_bytes,
                        a->out + (j*x + i)*out_bytes, out_bytes);
                }
            }
            STBI_FREE(a->out);
            image_data += img_len;
            image_data_len -= img_len;
        }
    }
    a->out = final;

    return 1;
}

static int stbi__compute_transparency(stbi__png *z, stbi_uc tc[3], int out_n)
{
    stbi__context *s = z->s;
    stbi__uint32 i, pixel_count = s->img_x * s->img_y;
    stbi_uc *p = z->out;

    // compute color-based transparency, assuming we've
    // already got 255 as the alpha value in the output
    STBI_ASSERT(out_n == 2 || out_n == 4);

    if (out_n == 2) {
        for (i = 0; i < pixel_count; ++i) {
            p[1] = (p[0] == tc[0] ? 0 : 255);
            p += 2;
        }
    }
    else {
        for (i = 0; i < pixel_count; ++i) {
            if (p[0] == tc[0] && p[1] == tc[1] && p[2] == tc[2])
                p[3] = 0;
            p += 4;
        }
    }
    return 1;
}

static int stbi__compute_transparency16(stbi__png *z, stbi__uint16 tc[3], int out_n)
{
    stbi__context *s = z->s;
    stbi__uint32 i, pixel_count = s->img_x * s->img_y;
    stbi__uint16 *p = (stbi__uint16*)z->out;

    // compute color-based transparency, assuming we've
    // already got 65535 as the alpha value in the output
    STBI_ASSERT(out_n == 2 || out_n == 4);

    if (out_n == 2) {
        for (i = 0; i < pixel_count; ++i) {
            p[1] = (p[0] == tc[0] ? 0 : 65535);
            p += 2;
        }
    }
    else {
        for (i = 0; i < pixel_count; ++i) {
            if (p[0] == tc[0] && p[1] == tc[1] && p[2] == tc[2])
                p[3] = 0;
            p += 4;
        }
    }
    return 1;
}

static int stbi__expand_png_palette(stbi__png *a, stbi_uc *palette, int len, int pal_img_n)
{
    stbi__uint32 i, pixel_count = a->s->img_x * a->s->img_y;
    stbi_uc *p, *temp_out, *orig = a->out;

    p = (stbi_uc *)stbi__malloc_mad2(pixel_count, pal_img_n, 0);
    if (p == NULL) return stbi__err("outofmem", "Out of memory");

    // between here and free(out) below, exitting would leak
    temp_out = p;

    if (pal_img_n == 3) {
        for (i = 0; i < pixel_count; ++i) {
            int n = orig[i] * 4;
            p[0] = palette[n];
            p[1] = palette[n + 1];
            p[2] = palette[n + 2];
            p += 3;
        }
    }
    else {
        for (i = 0; i < pixel_count; ++i) {
            int n = orig[i] * 4;
            p[0] = palette[n];
            p[1] = palette[n + 1];
            p[2] = palette[n + 2];
            p[3] = palette[n + 3];
            p += 4;
        }
    }
    STBI_FREE(a->out);
    a->out = temp_out;

    STBI_NOTUSED(len);

    return 1;
}

static int stbi__unpremultiply_on_load = 0;
static int stbi__de_iphone_flag = 0;

STBIDEF void stbi_set_unpremultiply_on_load(int flag_true_if_should_unpremultiply)
{
    stbi__unpremultiply_on_load = flag_true_if_should_unpremultiply;
}

STBIDEF void stbi_convert_iphone_png_to_rgb(int flag_true_if_should_convert)
{
    stbi__de_iphone_flag = flag_true_if_should_convert;
}

static void stbi__de_iphone(stbi__png *z)
{
    stbi__context *s = z->s;
    stbi__uint32 i, pixel_count = s->img_x * s->img_y;
    stbi_uc *p = z->out;

    if (s->img_out_n == 3) {  // convert bgr to rgb
        for (i = 0; i < pixel_count; ++i) {
            stbi_uc t = p[0];
            p[0] = p[2];
            p[2] = t;
            p += 3;
        }
    }
    else {
        STBI_ASSERT(s->img_out_n == 4);
        if (stbi__unpremultiply_on_load) {
            // convert bgr to rgb and unpremultiply
            for (i = 0; i < pixel_count; ++i) {
                stbi_uc a = p[3];
                stbi_uc t = p[0];
                if (a) {
                    p[0] = p[2] * 255 / a;
                    p[1] = p[1] * 255 / a;
                    p[2] = t * 255 / a;
                }
                else {
                    p[0] = p[2];
                    p[2] = t;
                }
                p += 4;
            }
        }
        else {
            // convert bgr to rgb
            for (i = 0; i < pixel_count; ++i) {
                stbi_uc t = p[0];
                p[0] = p[2];
                p[2] = t;
                p += 4;
            }
        }
    }
}

#define STBI__PNG_TYPE(a,b,c,d)  (((a) « 24) + ((b) « 16) + ((c) « 8) + (d))

static int stbi__parse_png_file(stbi__png *z, int scan, int req_comp)
{
    stbi_uc palette[1024], pal_img_n = 0;
    stbi_uc has_trans = 0, tc[3];
    stbi__uint16 tc16[3];
    stbi__uint32 ioff = 0, idata_limit = 0, i, pal_len = 0;
    int first = 1, k, interlace = 0, color = 0, is_iphone = 0;
    stbi__context *s = z->s;

    z->expanded = NULL;
    z->idata = NULL;
    z->out = NULL;

    if (!stbi__check_png_header(s)) return 0;

    if (scan == STBI__SCAN_type) return 1;

    for (;;) {
        stbi__pngchunk c = stbi__get_chunk_header(s);
        switch (c.type) {
        case STBI__PNG_TYPE('C', 'g', 'B', 'I'):
            is_iphone = 1;
            stbi__skip(s, c.length);
            break;
        case STBI__PNG_TYPE('I', 'H', 'D', 'R'): {
            int comp, filter;
            if (!first) return stbi__err("multiple IHDR", "Corrupt PNG");
            first = 0;
            if (c.length != 13) return stbi__err("bad IHDR len", "Corrupt PNG");
            s->img_x = stbi__get32be(s); if (s->img_x > (1 « 24)) return stbi__err("too large", "Very large image (corrupt?)");
            s->img_y = stbi__get32be(s); if (s->img_y > (1 « 24)) return stbi__err("too large", "Very large image (corrupt?)");
            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");
            color = stbi__get8(s);  if (color > 6)         return stbi__err("bad ctype", "Corrupt PNG");
            if (color == 3 && z->depth == 16)                  return stbi__err("bad ctype", "Corrupt PNG");
            if (color == 3) pal_img_n = 3; else if (color & 1) return stbi__err("bad ctype", "Corrupt PNG");
            comp = stbi__get8(s);  if (comp) return stbi__err("bad comp method", "Corrupt PNG");
            filter = stbi__get8(s);  if (filter) return stbi__err("bad filter method", "Corrupt PNG");
            interlace = stbi__get8(s); if (interlace>1) return stbi__err("bad interlace method", "Corrupt PNG");
            if (!s->img_x || !s->img_y) return stbi__err("0-pixel image", "Corrupt PNG");
            if (!pal_img_n) {
                s->img_n = (color & 2 ? 3 : 1) + (color & 4 ? 1 : 0);
                if ((1 « 30) / s->img_x / s->img_n < s->img_y) return stbi__err("too large", "Image too large to decode");
                if (scan == STBI__SCAN_header) return 1;
            }
            else {
                // if paletted, then pal_n is our final components, and
                // img_n is # components to decompress/filter.
                s->img_n = 1;
                if ((1 « 30) / s->img_x / 4 < s->img_y) return stbi__err("too large", "Corrupt PNG");
                // if SCAN_header, have to scan to see if we have a tRNS
            }
            break;
        }

        case STBI__PNG_TYPE('P', 'L', 'T', 'E'): {
            if (first) return stbi__err("first not IHDR", "Corrupt PNG");
            if (c.length > 256 * 3) return stbi__err("invalid PLTE", "Corrupt PNG");
            pal_len = c.length / 3;
            if (pal_len * 3 != c.length) return stbi__err("invalid PLTE", "Corrupt PNG");
            for (i = 0; i < pal_len; ++i) {
                palette[i * 4 + 0] = stbi__get8(s);
                palette[i * 4 + 1] = stbi__get8(s);
                palette[i * 4 + 2] = stbi__get8(s);
                palette[i * 4 + 3] = 255;
            }
            break;
        }

        case STBI__PNG_TYPE('t', 'R', 'N', 'S'): {
            if (first) return stbi__err("first not IHDR", "Corrupt PNG");
            if (z->idata) return stbi__err("tRNS after IDAT", "Corrupt PNG");
            if (pal_img_n) {
                if (scan == STBI__SCAN_header) { s->img_n = 4; return 1; }
                if (pal_len == 0) return stbi__err("tRNS before PLTE", "Corrupt PNG");
                if (c.length > pal_len) return stbi__err("bad tRNS len", "Corrupt PNG");
                pal_img_n = 4;
                for (i = 0; i < c.length; ++i)
                    palette[i * 4 + 3] = stbi__get8(s);
            }
            else {
                if (!(s->img_n & 1)) return stbi__err("tRNS with alpha", "Corrupt PNG");
                if (c.length != (stbi__uint32)s->img_n * 2) return stbi__err("bad tRNS len", "Corrupt PNG");
                has_trans = 1;
                if (z->depth == 16) {
                    for (k = 0; k < s->img_n; ++k) tc16[k] = (stbi__uint16)stbi__get16be(s); // copy the values as-is
                }
                else {
                    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
                }
            }
            break;
        }

        case STBI__PNG_TYPE('I', 'D', 'A', 'T'): {
            if (first) return stbi__err("first not IHDR", "Corrupt PNG");
            if (pal_img_n && !pal_len) return stbi__err("no PLTE", "Corrupt PNG");
            if (scan == STBI__SCAN_header) { s->img_n = pal_img_n; return 1; }
            if ((int)(ioff + c.length) < (int)ioff) return 0;
            if (ioff + c.length > idata_limit) {
                stbi__uint32 idata_limit_old = idata_limit;
                stbi_uc *p;
                if (idata_limit == 0) idata_limit = c.length > 4096 ? c.length : 4096;
                while (ioff + c.length > idata_limit)
                    idata_limit *= 2;
                STBI_NOTUSED(idata_limit_old);
                p = (stbi_uc *)STBI_REALLOC_SIZED(z->idata, idata_limit_old, idata_limit); if (p == NULL) return stbi__err("outofmem", "Out of memory");
                z->idata = p;
            }
            if (!stbi__getn(s, z->idata + ioff, c.length)) return stbi__err("outofdata", "Corrupt PNG");
            ioff += c.length;
            break;
        }

        case STBI__PNG_TYPE('I', 'E', 'N', 'D'): {
            stbi__uint32 raw_len, bpl;
            if (first) return stbi__err("first not IHDR", "Corrupt PNG");
            if (scan != STBI__SCAN_load) return 1;
            if (z->idata == NULL) return stbi__err("no IDAT", "Corrupt PNG");
            // initial guess for decoded data size to avoid unnecessary reallocs
            bpl = (s->img_x * z->depth + 7) / 8; // bytes per line, per component
            raw_len = bpl * s->img_y * s->img_n /* pixels */ + s->img_y /* filter mode per row */;
            z->expanded = (stbi_uc *)stbi_zlib_decode_malloc_guesssize_headerflag((char *)z->idata, ioff, raw_len, (int *)&raw_len, !is_iphone);
            if (z->expanded == NULL) return 0; // zlib should set error
            STBI_FREE(z->idata); z->idata = NULL;
            if ((req_comp == s->img_n + 1 && req_comp != 3 && !pal_img_n) || has_trans)
                s->img_out_n = s->img_n + 1;
            else
                s->img_out_n = s->img_n;
            if (!stbi__create_png_image(z, z->expanded, raw_len, s->img_out_n, z->depth, color, interlace)) return 0;
            if (has_trans) {
                if (z->depth == 16) {
                    if (!stbi__compute_transparency16(z, tc16, s->img_out_n)) return 0;
                }
                else {
                    if (!stbi__compute_transparency(z, tc, s->img_out_n)) return 0;
                }
            }
            if (is_iphone && stbi__de_iphone_flag && s->img_out_n > 2)
                stbi__de_iphone(z);
            if (pal_img_n) {
                // pal_img_n == 3 or 4
                s->img_n = pal_img_n; // record the actual colors we had
                s->img_out_n = pal_img_n;
                if (req_comp >= 3) s->img_out_n = req_comp;
                if (!stbi__expand_png_palette(z, palette, pal_len, s->img_out_n))
                    return 0;
            }
            STBI_FREE(z->expanded); z->expanded = NULL;
            return 1;
        }

        default:
            // if critical, fail
            if (first) return stbi__err("first not IHDR", "Corrupt PNG");
            if ((c.type & (1 « 29)) == 0) {
#ifndef STBI_NO_FAILURE_STRINGS
                // not threadsafe
                static char invalid_chunk[] = "XXXX PNG chunk not known";
                invalid_chunk[0] = STBI__BYTECAST(c.type » 24);
                invalid_chunk[1] = STBI__BYTECAST(c.type » 16);
                invalid_chunk[2] = STBI__BYTECAST(c.type » 8);
                invalid_chunk[3] = STBI__BYTECAST(c.type » 0);
#endif
                return stbi__err(invalid_chunk, "PNG not supported: unknown PNG chunk type");
            }
            stbi__skip(s, c.length);
            break;
        }
        // end of PNG chunk, read and skip CRC
        stbi__get32be(s);
    }
}

static void *stbi__do_png(stbi__png *p, int *x, int *y, int *n, int req_comp, stbi__result_info *ri)
{
    void *result = NULL;
    if (req_comp < 0 || req_comp > 4) return stbi__errpuc("bad req_comp", "Internal error");
    if (stbi__parse_png_file(p, STBI__SCAN_load, req_comp)) {
        if (p->depth < 8)
            ri->bits_per_channel = 8;
        else
            ri->bits_per_channel = p->depth;
        result = p->out;
        p->out = NULL;
        if (req_comp && req_comp != p->s->img_out_n) {
            if (ri->bits_per_channel == 8)
                result = stbi__convert_format((unsigned char *)result, p->s->img_out_n, req_comp,
  p->s->img_x, p->s->img_y);
            else
                result = stbi__convert_format16((stbi__uint16 *)result, p->s->img_out_n, req_comp,
  p->s->img_x, p->s->img_y);
            p->s->img_out_n = req_comp;
            if (result == NULL) return result;
        }
        *x = p->s->img_x;
        *y = p->s->img_y;
        if (n) *n = p->s->img_n;
    }
    STBI_FREE(p->out);      p->out = NULL;
    STBI_FREE(p->expanded); p->expanded = NULL;
    STBI_FREE(p->idata);    p->idata = NULL;

    return result;
}

static void *stbi__png_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
  stbi__result_info *ri)
{
    stbi__png p;
    p.s = s;
    return stbi__do_png(&p, x, y, comp, req_comp, ri);
}

static int stbi__png_test(stbi__context *s)
{
    int r;
    r = stbi__check_png_header(s);
    stbi__rewind(s);
    return r;
}

static int stbi__png_info_raw(stbi__png *p, int *x, int *y, int *comp)
{
    if (!stbi__parse_png_file(p, STBI__SCAN_header, 0)) {
        stbi__rewind(p->s);
        return 0;
    }
    if (x) *x = p->s->img_x;
    if (y) *y = p->s->img_y;
    if (comp) *comp = p->s->img_n;
    return 1;
}

static int stbi__png_info(stbi__context *s, int *x, int *y, int *comp)
{
    stbi__png p;
    p.s = s;
    return stbi__png_info_raw(&p, x, y, comp);
}
#endif

// Microsoft/Windows BMP image

#ifndef STBI_NO_BMP
static int stbi__bmp_test_raw(stbi__context *s)
{
    int r;
    int sz;
    if (stbi__get8(s) != 'B') return 0;
    if (stbi__get8(s) != 'M') return 0;
    stbi__get32le(s); // discard filesize
    stbi__get16le(s); // discard reserved
    stbi__get16le(s); // discard reserved
    stbi__get32le(s); // discard data offset
    sz = stbi__get32le(s);
    r = (sz == 12 || sz == 40 || sz == 56 || sz == 108 || sz == 124);
    return r;
}

static int stbi__bmp_test(stbi__context *s)
{
    int r = stbi__bmp_test_raw(s);
    stbi__rewind(s);
    return r;
}


// returns 0..31 for the highest set bit
static int stbi__high_bit(unsigned int z)
{
    int n = 0;
    if (z == 0) return -1;
    if (z >= 0x10000) n += 16, z »= 16;
    if (z >= 0x00100) n += 8, z »= 8;
    if (z >= 0x00010) n += 4, z »= 4;
    if (z >= 0x00004) n += 2, z »= 2;
    if (z >= 0x00002) n += 1, z »= 1;
    return n;
}

static int stbi__bitcount(unsigned int a)
{
    a = (a & 0x55555555) + ((a » 1) & 0x55555555); // max 2
    a = (a & 0x33333333) + ((a » 2) & 0x33333333); // max 4
    a = (a + (a » 4)) & 0x0f0f0f0f; // max 8 per 4, now 8 bits
    a = (a + (a » 8)); // max 16 per 8 bits
    a = (a + (a » 16)); // max 32 per 8 bits
    return a & 0xff;
}

static int stbi__shiftsigned(int v, int shift, int bits)
{
    int result;
    int z = 0;

    if (shift < 0) v «= -shift;
    else v »= shift;
    result = v;

    z = bits;
    while (z < 8) {
        result += v » z;
        z += bits;
    }
    return result;
}

typedef struct
{
    int bpp, offset, hsz;
    unsigned int mr, mg, mb, ma, all_a;
} stbi__bmp_data;

static void *stbi__bmp_parse_header(stbi__context *s, stbi__bmp_data *info)
{
    int hsz;
    if (stbi__get8(s) != 'B' || stbi__get8(s) != 'M') return stbi__errpuc("not BMP", "Corrupt BMP");
    stbi__get32le(s); // discard filesize
    stbi__get16le(s); // discard reserved
    stbi__get16le(s); // discard reserved
    info->offset = stbi__get32le(s);
    info->hsz = hsz = stbi__get32le(s);
    info->mr = info->mg = info->mb = info->ma = 0;

    if (hsz != 12 && hsz != 40 && hsz != 56 && hsz != 108 && hsz != 124) return stbi__errpuc("unknown
  BMP", "BMP type not supported: unknown");
    if (hsz == 12) {
        s->img_x = stbi__get16le(s);
        s->img_y = stbi__get16le(s);
    }
    else {
        s->img_x = stbi__get32le(s);
        s->img_y = stbi__get32le(s);
    }
    if (stbi__get16le(s) != 1) return stbi__errpuc("bad BMP", "bad BMP");
    info->bpp = stbi__get16le(s);
    if (info->bpp == 1) return stbi__errpuc("monochrome", "BMP type not supported: 1-bit");
    if (hsz != 12) {
        int compress = stbi__get32le(s);
        if (compress == 1 || compress == 2) return stbi__errpuc("BMP RLE", "BMP type not supported: RLE");
        stbi__get32le(s); // discard sizeof
        stbi__get32le(s); // discard hres
        stbi__get32le(s); // discard vres
        stbi__get32le(s); // discard colorsused
        stbi__get32le(s); // discard max important
        if (hsz == 40 || hsz == 56) {
            if (hsz == 56) {
                stbi__get32le(s);
                stbi__get32le(s);
                stbi__get32le(s);
                stbi__get32le(s);
            }
            if (info->bpp == 16 || info->bpp == 32) {
                if (compress == 0) {
                    if (info->bpp == 32) {
                        info->mr = 0xffu « 16;
                        info->mg = 0xffu « 8;
                        info->mb = 0xffu « 0;
                        info->ma = 0xffu « 24;
                        info->all_a = 0; // if all_a is 0 at end, then we loaded alpha channel but it was all 0
                    }
                    else {
                        info->mr = 31u « 10;
                        info->mg = 31u « 5;
                        info->mb = 31u « 0;
                    }
                }
                else if (compress == 3) {
                    info->mr = stbi__get32le(s);
                    info->mg = stbi__get32le(s);
                    info->mb = stbi__get32le(s);
                    // not documented, but generated by photoshop and handled by mspaint
                    if (info->mr == info->mg && info->mg == info->mb) {
                        // ?!?!?
                        return stbi__errpuc("bad BMP", "bad BMP");
                    }
                }
                else
                    return stbi__errpuc("bad BMP", "bad BMP");
            }
        }
        else {
            int i;
            if (hsz != 108 && hsz != 124)
                return stbi__errpuc("bad BMP", "bad BMP");
            info->mr = stbi__get32le(s);
            info->mg = stbi__get32le(s);
            info->mb = stbi__get32le(s);
            info->ma = stbi__get32le(s);
            stbi__get32le(s); // discard color space
            for (i = 0; i < 12; ++i)
                stbi__get32le(s); // discard color space parameters
            if (hsz == 124) {
                stbi__get32le(s); // discard rendering intent
                stbi__get32le(s); // discard offset of profile data
                stbi__get32le(s); // discard size of profile data
                stbi__get32le(s); // discard reserved
            }
        }
    }
    return (void *)1;
}


static void *stbi__bmp_load(stbi__context *s, int *x, int *y, int *comp, int req_comp, stbi__result_info *ri)
{
    stbi_uc *out;
    unsigned int mr = 0, mg = 0, mb = 0, ma = 0, all_a;
    stbi_uc pal[256][4];
    int psize = 0, i, j, width;
    int flip_vertically, pad, target;
    stbi__bmp_data info;
    STBI_NOTUSED(ri);

    info.all_a = 255;
    if (stbi__bmp_parse_header(s, &info) == NULL)
        return NULL; // error code already set

    flip_vertically = ((int)s->img_y) > 0;
    s->img_y = abs((int)s->img_y);

    mr = info.mr;
```

```
   mg = info.mg;
   mb = info.mb;
   ma = info.ma;
   all_a = info.all_a;

   if (info.hsz == 12) {
       if (info.bpp < 24)
           psize = (info.offset - 14 - 24) / 3;
   }
   else {
       if (info.bpp < 16)
           psize = (info.offset - 14 - info.hsz) » 2;
   }

   s->img_n = ma ? 4 : 3;
   if (req_comp && req_comp >= 3) // we can directly decode 3 or 4
       target = req_comp;
   else
       target = s->img_n; // if they want monochrome, we'll post-convert

                       // sanity-check size
   if (!stbi__mad3sizes_valid(target, s->img_x, s->img_y, 0))
       return stbi__errpuc("too large", "Corrupt BMP");

   out = (stbi_uc *)stbi__malloc_mad3(target, s->img_x, s->img_y, 0);
   if (!out) return stbi__errpuc("outofmem", "Out of memory");
   if (info.bpp < 16) {
       int z = 0;
       if (psize == 0 || psize > 256) { STBI_FREE(out); return stbi__errpuc("invalid", "Corrupt BMP");
 }
       for (i = 0; i < psize; ++i) {
           pal[i][2] = stbi__get8(s);
           pal[i][1] = stbi__get8(s);
           pal[i][0] = stbi__get8(s);
           if (info.hsz != 12) stbi__get8(s);
           pal[i][3] = 255;
       }
       stbi__skip(s, info.offset - 14 - info.hsz - psize * (info.hsz == 12 ? 3 : 4));
       if (info.bpp == 4) width = (s->img_x + 1) » 1;
       else if (info.bpp == 8) width = s->img_x;
       else { STBI_FREE(out); return stbi__errpuc("bad bpp", "Corrupt BMP"); }
       pad = (-width) & 3;
       for (j = 0; j < (int)s->img_y; ++j) {
           for (i = 0; i < (int)s->img_x; i += 2) {
               int v = stbi__get8(s), v2 = 0;
               if (info.bpp == 4) {
                   v2 = v & 15;
                   v »= 4;
               }
               out[z++] = pal[v][0];
               out[z++] = pal[v][1];
               out[z++] = pal[v][2];
               if (target == 4) out[z++] = 255;
               if (i + 1 == (int)s->img_x) break;
               v = (info.bpp == 8) ? stbi__get8(s) : v2;
               out[z++] = pal[v][0];
               out[z++] = pal[v][1];
               out[z++] = pal[v][2];
               if (target == 4) out[z++] = 255;
           }
           stbi__skip(s, pad);
       }
   }
   else {
       int rshift = 0, gshift = 0, bshift = 0, ashift = 0, rcount = 0, gcount = 0, bcount = 0, acount
 = 0;
       int z = 0;
       int easy = 0;
       stbi__skip(s, info.offset - 14 - info.hsz);
       if (info.bpp == 24) width = 3 * s->img_x;
       else if (info.bpp == 16) width = 2 * s->img_x;
       else /* bpp = 32 and pad = 0 */ width = 0;
       pad = (-width) & 3;
       if (info.bpp == 24) {
           easy = 1;
       }
       else if (info.bpp == 32) {
           if (mb == 0xff && mg == 0xff00 && mr == 0x00ff0000 && ma == 0xff000000)
               easy = 2;
       }
       if (!easy) {
           if (!mr || !mg || !mb) { STBI_FREE(out); return stbi__errpuc("bad masks", "Corrupt BMP"); }
           // right shift amt to put high bit in position #7
           rshift = stbi__high_bit(mr) - 7; rcount = stbi__bitcount(mr);
           gshift = stbi__high_bit(mg) - 7; gcount = stbi__bitcount(mg);
           bshift = stbi__high_bit(mb) - 7; bcount = stbi__bitcount(mb);
           ashift = stbi__high_bit(ma) - 7; acount = stbi__bitcount(ma);
```

```
        }
        for (j = 0; j < (int)s->img_y; ++j) {
            if (easy) {
                for (i = 0; i < (int)s->img_x; ++i) {
                    unsigned char a;
                    out[z + 2] = stbi__get8(s);
                    out[z + 1] = stbi__get8(s);
                    out[z + 0] = stbi__get8(s);
                    z += 3;
                    a = (easy == 2 ? stbi__get8(s) : 255);
                    all_a |= a;
                    if (target == 4) out[z++] = a;
                }
            }
            else {
                int bpp = info.bpp;
                for (i = 0; i < (int)s->img_x; ++i) {
                    stbi__uint32 v = (bpp == 16 ? (stbi__uint32)stbi__get16le(s) : stbi__get32le(s));
                    int a;
                    out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mr, rshift, rcount));
                    out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mg, gshift, gcount));
                    out[z++] = STBI__BYTECAST(stbi__shiftsigned(v & mb, bshift, bcount));
                    a = (ma ? stbi__shiftsigned(v & ma, ashift, acount) : 255);
                    all_a |= a;
                    if (target == 4) out[z++] = STBI__BYTECAST(a);
                }
            }
            stbi__skip(s, pad);
        }
    }

    // if alpha channel is all 0s, replace with all 255s
    if (target == 4 && all_a == 0)
        for (i = 4 * s->img_x*s->img_y - 1; i >= 0; i -= 4)
            out[i] = 255;

    if (flip_vertically) {
        stbi_uc t;
        for (j = 0; j < (int)s->img_y » 1; ++j) {
            stbi_uc *p1 = out + j     *s->img_x*target;
            stbi_uc *p2 = out + (s->img_y - 1 - j)*s->img_x*target;
            for (i = 0; i < (int)s->img_x*target; ++i) {
                t = p1[i], p1[i] = p2[i], p2[i] = t;
            }
        }
    }

    if (req_comp && req_comp != target) {
        out = stbi__convert_format(out, target, req_comp, s->img_x, s->img_y);
        if (out == NULL) return out; // stbi__convert_format frees input on failure
    }

    *x = s->img_x;
    *y = s->img_y;
    if (comp) *comp = s->img_n;
    return out;
}
#endif

// Targa Truevision - TGA
// by Jonathan Dummer
#ifndef STBI_NO_TGA
// returns STBI_rgb or whatever, 0 on error
static int stbi__tga_get_comp(int bits_per_pixel, int is_grey, int* is_rgb16)
{
    // only RGB or RGBA (incl. 16bit) or grey allowed
    if (is_rgb16) *is_rgb16 = 0;
    switch (bits_per_pixel) {
    case 8:  return STBI_grey;
    case 16: if (is_grey) return STBI_grey_alpha;
        // else: fall-through
    case 15: if (is_rgb16) *is_rgb16 = 1;
        return STBI_rgb;
    case 24: // fall-through
    case 32: return bits_per_pixel / 8;
    default: return 0;
    }
}

static int stbi__tga_info(stbi__context *s, int *x, int *y, int *comp)
{
    int tga_w, tga_h, tga_comp, tga_image_type, tga_bits_per_pixel, tga_colormap_bpp;
    int sz, tga_colormap_type;
    stbi__get8(s);                   // discard Offset
    tga_colormap_type = stbi__get8(s); // colormap type
    if (tga_colormap_type > 1) {
        stbi__rewind(s);
        return 0;      // only RGB or indexed allowed
    }
    tga_image_type = stbi__get8(s); // image type
    if (tga_colormap_type == 1) { // colormapped (paletted) image
        if (tga_image_type != 1 && tga_image_type != 9) {
            stbi__rewind(s);
            return 0;
        }
        stbi__skip(s, 4);       // skip index of first colormap entry and number of entries
        sz = stbi__get8(s);    //   check bits per palette color entry
        if ((sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32)) {
            stbi__rewind(s);
            return 0;
        }
        stbi__skip(s, 4);       // skip image x and y origin
        tga_colormap_bpp = sz;
    }
    else { // "normal" image w/o colormap - only RGB or grey allowed, +/- RLE
        if ((tga_image_type != 2) && (tga_image_type != 3) && (tga_image_type != 10) && (tga_image_type
 != 11)) {
            stbi__rewind(s);
            return 0; // only RGB or grey allowed, +/- RLE
        }
        stbi__skip(s, 9); // skip colormap specification and image x/y origin
        tga_colormap_bpp = 0;
    }
    tga_w = stbi__get16le(s);
    if (tga_w < 1) {
        stbi__rewind(s);
        return 0;   // test width
    }
    tga_h = stbi__get16le(s);
    if (tga_h < 1) {
        stbi__rewind(s);
        return 0;   // test height
    }
    tga_bits_per_pixel = stbi__get8(s); // bits per pixel
    stbi__get8(s); // ignore alpha bits
    if (tga_colormap_bpp != 0) {
        if ((tga_bits_per_pixel != 8) && (tga_bits_per_pixel != 16)) {
            // when using a colormap, tga_bits_per_pixel is the size of the indexes
            // I don't think anything but 8 or 16bit indexes makes sense
            stbi__rewind(s);
            return 0;
        }
        tga_comp = stbi__tga_get_comp(tga_colormap_bpp, 0, NULL);
    }
    else {
        tga_comp = stbi__tga_get_comp(tga_bits_per_pixel, (tga_image_type == 3) || (tga_image_type ==
 11), NULL);
    }
    if (!tga_comp) {
        stbi__rewind(s);
        return 0;
    }
    if (x) *x = tga_w;
    if (y) *y = tga_h;
    if (comp) *comp = tga_comp;
    return 1;                   // seems to have passed everything
}

static int stbi__tga_test(stbi__context *s)
{
    int res = 0;
    int sz, tga_color_type;
    stbi__get8(s);      //   discard Offset
    tga_color_type = stbi__get8(s);   //   color type
    if (tga_color_type > 1) goto errorEnd;   //   only RGB or indexed allowed
    sz = stbi__get8(s);   //   image type
    if (tga_color_type == 1) { // colormapped (paletted) image
        if (sz != 1 && sz != 9) goto errorEnd; // colortype 1 demands image type 1 or 9
        stbi__skip(s, 4);       // skip index of first colormap entry and number of entries
        sz = stbi__get8(s);    //   check bits per palette color entry
        if ((sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32)) goto errorEnd;
        stbi__skip(s, 4);       // skip image x and y origin
    }
    else { // "normal" image w/o colormap
        if ((sz != 2) && (sz != 3) && (sz != 10) && (sz != 11)) goto errorEnd; // only RGB or grey
 allowed, +/- RLE
        stbi__skip(s, 9); // skip colormap specification and image x/y origin
    }
    if (stbi__get16le(s) < 1) goto errorEnd;      //   test width
    if (stbi__get16le(s) < 1) goto errorEnd;      //   test height
    sz = stbi__get8(s);   //   bits per pixel
    if ((tga_color_type == 1) && (sz != 8) && (sz != 16)) goto errorEnd; // for colormapped images, bpp
 is size of an index
    if ((sz != 8) && (sz != 15) && (sz != 16) && (sz != 24) && (sz != 32)) goto errorEnd;

    res = 1; // if we got this far, everything's good and we can return 1 instead of 0

errorEnd:
    stbi__rewind(s);
    return res;
}

// read 16bit value and convert to 24bit RGB
static void stbi__tga_read_rgb16(stbi__context *s, stbi_uc* out)
{
    stbi__uint16 px = (stbi__uint16)stbi__get16le(s);
    stbi__uint16 fiveBitMask = 31;
    // we have 3 channels with 5bits each
    int r = (px » 10) & fiveBitMask;
    int g = (px » 5) & fiveBitMask;
    int b = px & fiveBitMask;
    // Note that this saves the data in RGB(A) order, so it doesn't need to be swapped later
    out[0] = (stbi_uc)((r * 255) / 31);
    out[1] = (stbi_uc)((g * 255) / 31);
    out[2] = (stbi_uc)((b * 255) / 31);

    // some people claim that the most significant bit might be used for alpha
    // (possibly if an alpha-bit is set in the "image descriptor byte")
    // but that only made 16bit test images completely translucent..
    // so let's treat all 15 and 16bit TGAs as RGB with no alpha.
}

static void *stbi__tga_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri)
{
    //   read in the TGA header stuff
    int tga_offset = stbi__get8(s);
    int tga_indexed = stbi__get8(s);
    int tga_image_type = stbi__get8(s);
    int tga_is_RLE = 0;
    int tga_palette_start = stbi__get16le(s);
    int tga_palette_len = stbi__get16le(s);
    int tga_palette_bits = stbi__get8(s);
    int tga_x_origin = stbi__get16le(s);
    int tga_y_origin = stbi__get16le(s);
    int tga_width = stbi__get16le(s);
    int tga_height = stbi__get16le(s);
    int tga_bits_per_pixel = stbi__get8(s);
    int tga_comp, tga_rgb16 = 0;
    int tga_inverted = stbi__get8(s);
    // int tga_alpha_bits = tga_inverted & 15; // the 4 lowest bits - unused (useless?)
    //   image data
    unsigned char *tga_data;
    unsigned char *tga_palette = NULL;
    int i, j;
    unsigned char raw_data[4] = { 0 };
    int RLE_count = 0;
    int RLE_repeating = 0;
    int read_next_pixel = 1;
    STBI_NOTUSED(ri);

    //   do a tiny bit of precessing
    if (tga_image_type >= 8)
    {
        tga_image_type -= 8;
        tga_is_RLE = 1;
    }
    tga_inverted = 1 - ((tga_inverted » 5) & 1);

    //   If I'm paletted, then I'll use the number of bits from the palette
    if (tga_indexed) tga_comp = stbi__tga_get_comp(tga_palette_bits, 0, &tga_rgb16);
    else tga_comp = stbi__tga_get_comp(tga_bits_per_pixel, (tga_image_type == 3), &tga_rgb16);

    if (!tga_comp) // shouldn't really happen, stbi__tga_test() should have ensured basic consistency
        return stbi__errpuc("bad format", "Can't find out TGA pixelformat");

    //   tga info
    *x = tga_width;
    *y = tga_height;
    if (comp) *comp = tga_comp;

    if (!stbi__mad3sizes_valid(tga_width, tga_height, tga_comp, 0))
        return stbi__errpuc("too large", "Corrupt TGA");

    tga_data = (unsigned char*)stbi__malloc_mad3(tga_width, tga_height, tga_comp, 0);
    if (!tga_data) return stbi__errpuc("outofmem", "Out of memory");

    // skip to the data's starting position (offset usually = 0)
    stbi__skip(s, tga_offset);

    if (!tga_indexed && !tga_is_RLE && !tga_rgb16) {
        for (i = 0; i < tga_height; ++i) {
            int row = tga_inverted ? tga_height - i - 1 : i;
            stbi_uc *tga_row = tga_data + row*tga_width*tga_comp;
            stbi__getn(s, tga_row, tga_width * tga_comp);
        }
    }
    else {
        //   do I need to load a palette?
        if (tga_indexed)
        {
            //   any data to skip? (offset usually = 0)
            stbi__skip(s, tga_palette_start);
            //   load the palette
            tga_palette = (unsigned char*)stbi__malloc_mad2(tga_palette_len, tga_comp, 0);
            if (!tga_palette) {
                STBI_FREE(tga_data);
                return stbi__errpuc("outofmem", "Out of memory");
            }
            if (tga_rgb16) {
                stbi_uc *pal_entry = tga_palette;
                STBI_ASSERT(tga_comp == STBI_rgb);
                for (i = 0; i < tga_palette_len; ++i) {
                    stbi__tga_read_rgb16(s, pal_entry);
                    pal_entry += tga_comp;
                }
            }
            else if (!stbi__getn(s, tga_palette, tga_palette_len * tga_comp)) {
                STBI_FREE(tga_data);
                STBI_FREE(tga_palette);
                return stbi__errpuc("bad palette", "Corrupt TGA");
            }
        }
        //   load the data
        for (i = 0; i < tga_width * tga_height; ++i)
        {
            //   if I'm in RLE mode, do I need to get a RLE stbi__pngchunk?
            if (tga_is_RLE)
            {
                if (RLE_count == 0)
                {
                    //   yep, get the next byte as a RLE command
                    int RLE_cmd = stbi__get8(s);
                    RLE_count = 1 + (RLE_cmd & 127);
                    RLE_repeating = RLE_cmd » 7;
                    read_next_pixel = 1;
                }
                else if (!RLE_repeating)
                {
                    read_next_pixel = 1;
                }
            }
            else
            {
                read_next_pixel = 1;
            }
            //   OK, if I need to read a pixel, do it now
            if (read_next_pixel)
            {
                //   load however much data we did have
                if (tga_indexed)
                {
                    // read in index, then perform the lookup
                    int pal_idx = (tga_bits_per_pixel == 8) ? stbi__get8(s) : stbi__get16le(s);
                    if (pal_idx >= tga_palette_len) {
                        // invalid index
                        pal_idx = 0;
                    }
                    pal_idx *= tga_comp;
                    for (j = 0; j < tga_comp; ++j) {
                        raw_data[j] = tga_palette[pal_idx + j];
                    }
                }
                else if (tga_rgb16) {
                    STBI_ASSERT(tga_comp == STBI_rgb);
                    stbi__tga_read_rgb16(s, raw_data);
                }
                else {
                    //   read in the data raw
                    for (j = 0; j < tga_comp; ++j) {
                        raw_data[j] = stbi__get8(s);
                    }
                }
                //   clear the reading flag for the next pixel
                read_next_pixel = 0;
            } // end of reading a pixel

              // copy data
```

```
            for (j = 0; j < tga_comp; ++j)
                tga_data[i*tga_comp + j] = raw_data[j];

            //   in case we're in RLE mode, keep counting down
            --RLE_count;
        }
        //   do I need to invert the image?
        if (tga_inverted)
        {
            for (j = 0; j * 2 < tga_height; ++j)
            {
                int index1 = j * tga_width * tga_comp;
                int index2 = (tga_height - 1 - j) * tga_width * tga_comp;
                for (i = tga_width * tga_comp; i > 0; --i)
                {
                    unsigned char temp = tga_data[index1];
                    tga_data[index1] = tga_data[index2];
                    tga_data[index2] = temp;
                    ++index1;
                    ++index2;
                }
            }
        }
        //   clear my palette, if I had one
        if (tga_palette != NULL)
        {
            STBI_FREE(tga_palette);
        }
    }

    // swap RGB - if the source data was RGB16, it already is in the right order
    if (tga_comp >= 3 && !tga_rgb16)
    {
        unsigned char* tga_pixel = tga_data;
        for (i = 0; i < tga_width * tga_height; ++i)
        {
            unsigned char temp = tga_pixel[0];
            tga_pixel[0] = tga_pixel[2];
            tga_pixel[2] = temp;
            tga_pixel += tga_comp;
        }
    }

    // convert to target component count
    if (req_comp && req_comp != tga_comp)
        tga_data = stbi__convert_format(tga_data, tga_comp, req_comp, tga_width, tga_height);

    //   the things I do to get rid of an error message, and yet keep
    //   Microsoft's C compilers happy... [8^(
    tga_palette_start = tga_palette_len = tga_palette_bits =
        tga_x_origin = tga_y_origin = 0;
    //   OK, done
    return tga_data;
}
#endif

// *************************************************************************************************
// Photoshop PSD loader -- PD by Thatcher Ulrich, integration by Nicolas Schulz, tweaked by STB

#ifndef STBI_NO_PSD
static int stbi__psd_test(stbi__context *s)
{
    int r = (stbi__get32be(s) == 0x38425053);
    stbi__rewind(s);
    return r;
}

static int stbi__psd_decode_rle(stbi__context *s, stbi_uc *p, int pixelCount)
{
    int count, nleft, len;

    count = 0;
    while ((nleft = pixelCount - count) > 0) {
        len = stbi__get8(s);
        if (len == 128) {
            // No-op.
        }
        else if (len < 128) {
            // Copy next len+1 bytes literally.
            len++;
            if (len > nleft) return 0; // corrupt data
            count += len;
            while (len) {
                *p = stbi__get8(s);
                p += 4;
                len--;
            }
        }
        else if (len > 128) {
            stbi_uc   val;
            // Next -len+1 bytes in the dest are replicated from next source byte.
            // (Interpret len as a negative 8-bit int.)
            len = 257 - len;
            if (len > nleft) return 0; // corrupt data
            val = stbi__get8(s);
            count += len;
            while (len) {
                *p = val;
                p += 4;
                len--;
            }
        }
    }

    return 1;
}

static void *stbi__psd_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
  stbi__result_info *ri, int bpc)
{
    int pixelCount;
    int channelCount, compression;
    int channel, i;
    int bitdepth;
    int w, h;
    stbi_uc *out;
    STBI_NOTUSED(ri);

    // Check identifier
    if (stbi__get32be(s) != 0x38425053)   // "8BPS"
        return stbi__errpuc("not PSD", "Corrupt PSD image");

    // Check file type version.
    if (stbi__get16be(s) != 1)
        return stbi__errpuc("wrong version", "Unsupported version of PSD image");

    // Skip 6 reserved bytes.
    stbi__skip(s, 6);

    // Read the number of channels (R, G, B, A, etc).
    channelCount = stbi__get16be(s);
    if (channelCount < 0 || channelCount > 16)
        return stbi__errpuc("wrong channel count", "Unsupported number of channels in PSD image");

    // Read the rows and columns of the image.
    h = stbi__get32be(s);
    w = stbi__get32be(s);

    // Make sure the depth is 8 bits.
    bitdepth = stbi__get16be(s);
    if (bitdepth != 8 && bitdepth != 16)
        return stbi__errpuc("unsupported bit depth", "PSD bit depth is not 8 or 16 bit");

    // Make sure the color mode is RGB.
    // Valid options are:
    //   0: Bitmap
    //   1: Grayscale
    //   2: Indexed color
    //   3: RGB color
    //   4: CMYK color
    //   7: Multichannel
    //   8: Duotone
    //   9: Lab color
    if (stbi__get16be(s) != 3)
        return stbi__errpuc("wrong color format", "PSD is not in RGB color format");

    // Skip the Mode Data.  (It's the palette for indexed color; other info for other modes.)
    stbi__skip(s, stbi__get32be(s));

    // Skip the image resources.  (resolution, pen tool paths, etc)
    stbi__skip(s, stbi__get32be(s));

    // Skip the reserved data.
    stbi__skip(s, stbi__get32be(s));

    // Find out if the data is compressed.
    // Known values:
    //   0: no compression
    //   1: RLE compressed
    compression = stbi__get16be(s);
    if (compression > 1)
        return stbi__errpuc("bad compression", "PSD has an unknown compression format");

    // Check size
    if (!stbi__mad3sizes_valid(4, w, h, 0))
        return stbi__errpuc("too large", "Corrupt PSD");

    // Create the destination image.

    if (!compression && bitdepth == 16 && bpc == 16) {
        out = (stbi_uc *)stbi__malloc_mad3(8, w, h, 0);
        ri->bits_per_channel = 16;
    }
    else
        out = (stbi_uc *)stbi__malloc(4 * w*h);

    if (!out) return stbi__errpuc("outofmem", "Out of memory");
    pixelCount = w*h;

    // Initialize the data to zero.
    //memset( out, 0, pixelCount * 4 );

    // Finally, the image data.
    if (compression) {
        // RLE as used by .PSD and .TIFF
        // Loop until you get the number of unpacked bytes you are expecting:
        //     Read the next source byte into n.
        //     If n is between 0 and 127 inclusive, copy the next n+1 bytes literally.
        //     Else if n is between -127 and -1 inclusive, copy the next byte -n+1 times.
        //     Else if n is 128, noop.
        // Endloop

        // The RLE-compressed data is preceeded by a 2-byte data count for each row in the data,
        // which we're going to just skip.
        stbi__skip(s, h * channelCount * 2);

        // Read the RLE data by channel.
        for (channel = 0; channel < 4; channel++) {
            stbi_uc *p;

            p = out + channel;
            if (channel >= channelCount) {
                // Fill this channel with default data.
                for (i = 0; i < pixelCount; i++, p += 4)
                    *p = (channel == 3 ? 255 : 0);
            }
            else {
                // Read the RLE data.
                if (!stbi__psd_decode_rle(s, p, pixelCount)) {
                    STBI_FREE(out);
                    return stbi__errpuc("corrupt", "bad RLE data");
                }
            }
        }

    }
    else {
        // We're at the raw image data.  It's each channel in order (Red, Green, Blue, Alpha, ...)
        // where each channel consists of an 8-bit (or 16-bit) value for each pixel in the image.

        // Read the data by channel.
        for (channel = 0; channel < 4; channel++) {
            if (channel >= channelCount) {
                // Fill this channel with default data.
                if (bitdepth == 16 && bpc == 16) {
                    stbi__uint16 *q = ((stbi__uint16 *)out) + channel;
                    stbi__uint16 val = channel == 3 ? 65535 : 0;
                    for (i = 0; i < pixelCount; i++, q += 4)
                        *q = val;
                }
                else {
                    stbi_uc *p = out + channel;
                    stbi_uc val = channel == 3 ? 255 : 0;
                    for (i = 0; i < pixelCount; i++, p += 4)
                        *p = val;
                }
            }
            else {
                if (ri->bits_per_channel == 16) {    // output bpc
                    stbi__uint16 *q = ((stbi__uint16 *)out) + channel;
                    for (i = 0; i < pixelCount; i++, q += 4)
                        *q = (stbi__uint16)stbi__get16be(s);
                }
                else {
                    stbi_uc *p = out + channel;
                    if (bitdepth == 16) {  // input bpc
                        for (i = 0; i < pixelCount; i++, p += 4)
                            *p = (stbi_uc)(stbi__get16be(s) » 8);
                    }
                    else {
                        for (i = 0; i < pixelCount; i++, p += 4)
```

```
                              *p = stbi__get8(s);
                        }
                    }
                }
            }
    }

    // remove weird white matte from PSD
    if (channelCount >= 4) {
        if (ri->bits_per_channel == 16) {
            for (i = 0; i < w*h; ++i) {
                stbi__uint16 *pixel = (stbi__uint16 *)out + 4 * i;
                if (pixel[3] != 0 && pixel[3] != 65535) {
                    float a = pixel[3] / 65535.0f;
                    float ra = 1.0f / a;
                    float inv_a = 65535.0f * (1 - ra);
                    pixel[0] = (stbi__uint16)(pixel[0] * ra + inv_a);
                    pixel[1] = (stbi__uint16)(pixel[1] * ra + inv_a);
                    pixel[2] = (stbi__uint16)(pixel[2] * ra + inv_a);
                }
            }
        }
        else {
            for (i = 0; i < w*h; ++i) {
                unsigned char *pixel = out + 4 * i;
                if (pixel[3] != 0 && pixel[3] != 255) {
                    float a = pixel[3] / 255.0f;
                    float ra = 1.0f / a;
                    float inv_a = 255.0f * (1 - ra);
                    pixel[0] = (unsigned char)(pixel[0] * ra + inv_a);
                    pixel[1] = (unsigned char)(pixel[1] * ra + inv_a);
                    pixel[2] = (unsigned char)(pixel[2] * ra + inv_a);
                }
            }
        }
    }

    // convert to desired output format
    if (req_comp && req_comp != 4) {
        if (ri->bits_per_channel == 16)
            out = (stbi_uc *)stbi__convert_format16((stbi__uint16 *)out, 4, req_comp, w, h);
        else
            out = stbi__convert_format(out, 4, req_comp, w, h);
        if (out == NULL) return out; // stbi__convert_format frees input on failure
    }

    if (comp) *comp = 4;
    *y = h;
    *x = w;

    return out;
}
#endif

// *************************************************************************************************
// Softimage PIC loader
// by Tom Seddon
//
// See http://softimage.wiki.softimage.com/index.php/INFO:_PIC_file_format
// See http://ozviz.wasp.uwa.edu.au/~pbourke/dataformats/softimagepic/

#ifndef STBI_NO_PIC
static int stbi__pic_is4(stbi__context *s, const char *str)
{
    int i;
    for (i = 0; i<4; ++i)
        if (stbi__get8(s) != (stbi_uc)str[i])
            return 0;

    return 1;
}

static int stbi__pic_test_core(stbi__context *s)
{
    int i;

    if (!stbi__pic_is4(s, "\x53\x80\xF6\x34"))
        return 0;

    for (i = 0; i<84; ++i)
        stbi__get8(s);

    if (!stbi__pic_is4(s, "PICT"))
        return 0;

    return 1;
}
```

```c

typedef struct
{
    stbi_uc size, type, channel;
} stbi__pic_packet;

static stbi_uc *stbi__readval(stbi__context *s, int channel, stbi_uc *dest)
{
    int mask = 0x80, i;

    for (i = 0; i<4; ++i, mask >>= 1) {
        if (channel & mask) {
            if (stbi__at_eof(s)) return stbi__errpuc("bad file", "PIC file too short");
            dest[i] = stbi__get8(s);
        }
    }

    return dest;
}

static void stbi__copyval(int channel, stbi_uc *dest, const stbi_uc *src)
{
    int mask = 0x80, i;

    for (i = 0; i<4; ++i, mask >>= 1)
        if (channel&mask)
            dest[i] = src[i];
}

static stbi_uc *stbi__pic_load_core(stbi__context *s, int width, int height, int *comp, stbi_uc
  *result)
{
    int act_comp = 0, num_packets = 0, y, chained;
    stbi__pic_packet packets[10];

    // this will (should...) cater for even some bizarre stuff like having data
    // for the same channel in multiple packets.
    do {
        stbi__pic_packet *packet;

        if (num_packets == sizeof(packets) / sizeof(packets[0]))
            return stbi__errpuc("bad format", "too many packets");

        packet = &packets[num_packets++];

        chained = stbi__get8(s);
        packet->size = stbi__get8(s);
        packet->type = stbi__get8(s);
        packet->channel = stbi__get8(s);

        act_comp |= packet->channel;

        if (stbi__at_eof(s))          return stbi__errpuc("bad file", "file too short (reading
  packets)");
        if (packet->size != 8)  return stbi__errpuc("bad format", "packet isn't 8bpp");
    } while (chained);

    *comp = (act_comp & 0x10 ? 4 : 3); // has alpha channel?

    for (y = 0; y<height; ++y) {
        int packet_idx;

        for (packet_idx = 0; packet_idx < num_packets; ++packet_idx) {
            stbi__pic_packet *packet = &packets[packet_idx];
            stbi_uc *dest = result + y*width * 4;

            switch (packet->type) {
            default:
                return stbi__errpuc("bad format", "packet has bad compression type");

            case 0: {//uncompressed
                int x;

                for (x = 0; x<width; ++x, dest += 4)
                    if (!stbi__readval(s, packet->channel, dest))
                        return 0;
                break;
            }

            case 1://Pure RLE
            {
                int left = width, i;

                while (left>0) {
                    stbi_uc count, value[4];

                    count = stbi__get8(s);
```

```
                     if (stbi__at_eof(s))   return stbi__errpuc("bad file", "file too short (pure read
   count)");

                     if (count > left)
                         count = (stbi_uc)left;

                     if (!stbi__readval(s, packet->channel, value))  return 0;

                     for (i = 0; i<count; ++i, dest += 4)
                         stbi__copyval(packet->channel, dest, value);
                     left -= count;
                 }
             }
             break;

             case 2: {//Mixed RLE
                 int left = width;
                 while (left>0) {
                     int count = stbi__get8(s), i;
                     if (stbi__at_eof(s))  return stbi__errpuc("bad file", "file too short (mixed read
   count)");

                     if (count >= 128) { // Repeated
                         stbi_uc value[4];

                         if (count == 128)
                             count = stbi__get16be(s);
                         else
                             count -= 127;
                         if (count > left)
                             return stbi__errpuc("bad file", "scanline overrun");

                         if (!stbi__readval(s, packet->channel, value))
                             return 0;

                         for (i = 0; i<count; ++i, dest += 4)
                             stbi__copyval(packet->channel, dest, value);
                     }
                     else { // Raw
                         ++count;
                         if (count>left) return stbi__errpuc("bad file", "scanline overrun");

                         for (i = 0; i<count; ++i, dest += 4)
                             if (!stbi__readval(s, packet->channel, dest))
                                 return 0;
                     }
                     left -= count;
                 }
                 break;
             }
             }
         }
     }

     return result;
 }

 static void *stbi__pic_load(stbi__context *s, int *px, int *py, int *comp, int req_comp,
   stbi__result_info *ri)
 {
     stbi_uc *result;
     int i, x, y;
     STBI_NOTUSED(ri);

     for (i = 0; i<92; ++i)
         stbi__get8(s);

     x = stbi__get16be(s);
     y = stbi__get16be(s);
     if (stbi__at_eof(s))  return stbi__errpuc("bad file", "file too short (pic header)");
     if (!stbi__mad3sizes_valid(x, y, 4, 0)) return stbi__errpuc("too large", "PIC image too large to
   decode");

     stbi__get32be(s); //skip `ratio'
     stbi__get16be(s); //skip `fields'
     stbi__get16be(s); //skip `pad'

                     // intermediate buffer is RGBA
     result = (stbi_uc *)stbi__malloc_mad3(x, y, 4, 0);
     memset(result, 0xff, x*y * 4);

     if (!stbi__pic_load_core(s, x, y, comp, result)) {
         STBI_FREE(result);
         result = 0;
     }
     *px = x;
     *py = y;
```

```
    if (req_comp == 0) req_comp = *comp;
    result = stbi__convert_format(result, 4, req_comp, x, y);

    return result;
}

static int stbi__pic_test(stbi__context *s)
{
    int r = stbi__pic_test_core(s);
    stbi__rewind(s);
    return r;
}
#endif

// *************************************************************************************************
// GIF loader -- public domain by Jean-Marc Lienher -- simplified/shrunk by stb

#ifndef STBI_NO_GIF
typedef struct
{
    stbi__int16 prefix;
    stbi_uc first;
    stbi_uc suffix;
} stbi__gif_lzw;

typedef struct
{
    int w, h;
    stbi_uc *out, *old_out;             // output buffer (always 4 components)
    int flags, bgindex, ratio, transparent, eflags, delay;
    stbi_uc  pal[256][4];
    stbi_uc lpal[256][4];
    stbi__gif_lzw codes[4096];
    stbi_uc *color_table;
    int parse, step;
    int lflags;
    int start_x, start_y;
    int max_x, max_y;
    int cur_x, cur_y;
    int line_size;
} stbi__gif;

static int stbi__gif_test_raw(stbi__context *s)
{
    int sz;
    if (stbi__get8(s) != 'G' || stbi__get8(s) != 'I' || stbi__get8(s) != 'F' || stbi__get8(s) != '8')
  return 0;
    sz = stbi__get8(s);
    if (sz != '9' && sz != '7') return 0;
    if (stbi__get8(s) != 'a') return 0;
    return 1;
}

static int stbi__gif_test(stbi__context *s)
{
    int r = stbi__gif_test_raw(s);
    stbi__rewind(s);
    return r;
}

static void stbi__gif_parse_colortable(stbi__context *s, stbi_uc pal[256][4], int num_entries, int
  transp)
{
    int i;
    for (i = 0; i < num_entries; ++i) {
        pal[i][2] = stbi__get8(s);
        pal[i][1] = stbi__get8(s);
        pal[i][0] = stbi__get8(s);
        pal[i][3] = transp == i ? 0 : 255;
    }
}

static int stbi__gif_header(stbi__context *s, stbi__gif *g, int *comp, int is_info)
{
    stbi_uc version;
    if (stbi__get8(s) != 'G' || stbi__get8(s) != 'I' || stbi__get8(s) != 'F' || stbi__get8(s) != '8')
        return stbi__err("not GIF", "Corrupt GIF");

    version = stbi__get8(s);
    if (version != '7' && version != '9')    return stbi__err("not GIF", "Corrupt GIF");
    if (stbi__get8(s) != 'a')                return stbi__err("not GIF", "Corrupt GIF");

    stbi__g_failure_reason = "";
    g->w = stbi__get16le(s);
    g->h = stbi__get16le(s);
    g->flags = stbi__get8(s);
    g->bgindex = stbi__get8(s);
    g->ratio = stbi__get8(s);
    g->transparent = -1;

    if (comp != 0) *comp = 4;  // can't actually tell whether it's 3 or 4 until we parse the comments

    if (is_info) return 1;

    if (g->flags & 0x80)
        stbi__gif_parse_colortable(s, g->pal, 2 « (g->flags & 7), -1);

    return 1;
}

static int stbi__gif_info_raw(stbi__context *s, int *x, int *y, int *comp)
{
    stbi__gif* g = (stbi__gif*)stbi__malloc(sizeof(stbi__gif));
    if (!stbi__gif_header(s, g, comp, 1)) {
        STBI_FREE(g);
        stbi__rewind(s);
        return 0;
    }
    if (x) *x = g->w;
    if (y) *y = g->h;
    STBI_FREE(g);
    return 1;
}

static void stbi__out_gif_code(stbi__gif *g, stbi__uint16 code)
{
    stbi_uc *p, *c;

    // recurse to decode the prefixes, since the linked-list is backwards,
    // and working backwards through an interleaved image would be nasty
    if (g->codes[code].prefix >= 0)
        stbi__out_gif_code(g, g->codes[code].prefix);

    if (g->cur_y >= g->max_y) return;

    p = &g->out[g->cur_x + g->cur_y];
    c = &g->color_table[g->codes[code].suffix * 4];

    if (c[3] >= 128) {
        p[0] = c[2];
        p[1] = c[1];
        p[2] = c[0];
        p[3] = c[3];
    }
    g->cur_x += 4;

    if (g->cur_x >= g->max_x) {
        g->cur_x = g->start_x;
        g->cur_y += g->step;

        while (g->cur_y >= g->max_y && g->parse > 0) {
            g->step = (1 « g->parse) * g->line_size;
            g->cur_y = g->start_y + (g->step » 1);
            --g->parse;
        }
    }
}

static stbi_uc *stbi__process_gif_raster(stbi__context *s, stbi__gif *g)
{
    stbi_uc lzw_cs;
    stbi__int32 len, init_code;
    stbi__uint32 first;
    stbi__int32 codesize, codemask, avail, oldcode, bits, valid_bits, clear;
    stbi__gif_lzw *p;

    lzw_cs = stbi__get8(s);
    if (lzw_cs > 12) return NULL;
    clear = 1 « lzw_cs;
    first = 1;
    codesize = lzw_cs + 1;
    codemask = (1 « codesize) - 1;
    bits = 0;
    valid_bits = 0;
    for (init_code = 0; init_code < clear; init_code++) {
        g->codes[init_code].prefix = -1;
        g->codes[init_code].first = (stbi_uc)init_code;
        g->codes[init_code].suffix = (stbi_uc)init_code;
    }

    // support no starting clear code
    avail = clear + 2;
    oldcode = -1;

    len = 0;
    for (;;) {
        if (valid_bits < codesize) {
            if (len == 0) {
                len = stbi__get8(s); // start new block
                if (len == 0)
                    return g->out;
            }
            --len;
            bits |= (stbi__int32)stbi__get8(s) « valid_bits;
            valid_bits += 8;
        }
        else {
            stbi__int32 code = bits & codemask;
            bits »= codesize;
            valid_bits -= codesize;
            // @OPTIMIZE: is there some way we can accelerate the non-clear path?
            if (code == clear) {  // clear code
                codesize = lzw_cs + 1;
                codemask = (1 « codesize) - 1;
                avail = clear + 2;
                oldcode = -1;
                first = 0;
            }
            else if (code == clear + 1) { // end of stream code
                stbi__skip(s, len);
                while ((len = stbi__get8(s)) > 0)
                    stbi__skip(s, len);
                return g->out;
            }
            else if (code <= avail) {
                if (first) return stbi__errpuc("no clear code", "Corrupt GIF");

                if (oldcode >= 0) {
                    p = &g->codes[avail++];
                    if (avail > 4096)        return stbi__errpuc("too many codes", "Corrupt GIF");
                    p->prefix = (stbi__int16)oldcode;
                    p->first = g->codes[oldcode].first;
                    p->suffix = (code == avail) ? p->first : g->codes[code].first;
                }
                else if (code == avail)
                    return stbi__errpuc("illegal code in raster", "Corrupt GIF");

                stbi__out_gif_code(g, (stbi__uint16)code);

                if ((avail & codemask) == 0 && avail <= 0x0FFF) {
                    codesize++;
                    codemask = (1 « codesize) - 1;
                }

                oldcode = code;
            }
            else {
                return stbi__errpuc("illegal code in raster", "Corrupt GIF");
            }
        }
    }
}

static void stbi__fill_gif_background(stbi__gif *g, int x0, int y0, int x1, int y1)
{
    int x, y;
    stbi_uc *c = g->pal[g->bgindex];
    for (y = y0; y < y1; y += 4 * g->w) {
        for (x = x0; x < x1; x += 4) {
            stbi_uc *p = &g->out[y + x];
            p[0] = c[2];
            p[1] = c[1];
            p[2] = c[0];
            p[3] = 0;
        }
    }
}

// this function is designed to support animated gifs, although stb_image doesn't support it
static stbi_uc *stbi__gif_load_next(stbi__context *s, stbi__gif *g, int *comp, int req_comp)
{
    int i;
    stbi_uc *prev_out = 0;

    if (g->out == 0 && !stbi__gif_header(s, g, comp, 0))
        return 0; // stbi__g_failure_reason set by stbi__gif_header

    if (!stbi__mad3sizes_valid(g->w, g->h, 4, 0))
        return stbi__errpuc("too large", "GIF too large");

    prev_out = g->out;
    g->out = (stbi_uc *)stbi__malloc_mad3(4, g->w, g->h, 0);
    if (g->out == 0) return stbi__errpuc("outofmem", "Out of memory");

    switch ((g->eflags & 0x1C) » 2) {
    case 0: // unspecified (also always used on 1st frame)
        stbi__fill_gif_background(g, 0, 0, 4 * g->w, 4 * g->w * g->h);
        break;
    case 1: // do not dispose
        if (prev_out) memcpy(g->out, prev_out, 4 * g->w * g->h);
        g->old_out = prev_out;
        break;
    case 2: // dispose to background
        if (prev_out) memcpy(g->out, prev_out, 4 * g->w * g->h);
        stbi__fill_gif_background(g, g->start_x, g->start_y, g->max_x, g->max_y);
        break;
    case 3: // dispose to previous
        if (g->old_out) {
            for (i = g->start_y; i < g->max_y; i += 4 * g->w)
                memcpy(&g->out[i + g->start_x], &g->old_out[i + g->start_x], g->max_x - g->start_x);
        }
        break;
    }

    for (;;) {
        switch (stbi__get8(s)) {
        case 0x2C: /* Image Descriptor */
        {
            int prev_trans = -1;
            stbi__int32 x, y, w, h;
            stbi_uc *o;

            x = stbi__get16le(s);
            y = stbi__get16le(s);
            w = stbi__get16le(s);
            h = stbi__get16le(s);
            if (((x + w) > (g->w)) || ((y + h) > (g->h)))
                return stbi__errpuc("bad Image Descriptor", "Corrupt GIF");

            g->line_size = g->w * 4;
            g->start_x = x * 4;
            g->start_y = y * g->line_size;
            g->max_x = g->start_x + w * 4;
            g->max_y = g->start_y + h * g->line_size;
            g->cur_x = g->start_x;
            g->cur_y = g->start_y;

            g->lflags = stbi__get8(s);

            if (g->lflags & 0x40) {
                g->step = 8 * g->line_size; // first interlaced spacing
                g->parse = 3;
            }
            else {
                g->step = g->line_size;
                g->parse = 0;
            }

            if (g->lflags & 0x80) {
                stbi__gif_parse_colortable(s, g->lpal, 2 « (g->lflags & 7), g->eflags & 0x01 ?
  g->transparent : -1);
                g->color_table = (stbi_uc *)g->lpal;
            }
            else if (g->flags & 0x80) {
                if (g->transparent >= 0 && (g->eflags & 0x01)) {
                    prev_trans = g->pal[g->transparent][3];
                    g->pal[g->transparent][3] = 0;
                }
                g->color_table = (stbi_uc *)g->pal;
            }
            else
                return stbi__errpuc("missing color table", "Corrupt GIF");

            o = stbi__process_gif_raster(s, g);
            if (o == NULL) return NULL;

            if (prev_trans != -1)
                g->pal[g->transparent][3] = (stbi_uc)prev_trans;

            return o;
        }

        case 0x21: // Comment Extension.
        {
            int len;
            if (stbi__get8(s) == 0xF9) { // Graphic Control Extension.
                len = stbi__get8(s);
                if (len == 4) {
                    g->eflags = stbi__get8(s);
                    g->delay = stbi__get16le(s);
                    g->transparent = stbi__get8(s);
                }
                else {
                    stbi__skip(s, len);
                    break;
                }
            }
            while ((len = stbi__get8(s)) != 0)
                stbi__skip(s, len);
            break;
        }

        case 0x3B: // gif stream termination code
            return (stbi_uc *)s; // using '1' causes warning on some compilers

        default:
            return stbi__errpuc("unknown code", "Corrupt GIF");
        }
    }

    STBI_NOTUSED(req_comp);
}

static void *stbi__gif_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
  stbi__result_info *ri)
{
    stbi_uc *u = 0;
    stbi__gif* g = (stbi__gif*)stbi__malloc(sizeof(stbi__gif));
    memset(g, 0, sizeof(*g));
    STBI_NOTUSED(ri);

    u = stbi__gif_load_next(s, g, comp, req_comp);
    if (u == (stbi_uc *)s) u = 0;  // end of animated gif marker
    if (u) {
        *x = g->w;
        *y = g->h;
        if (req_comp && req_comp != 4)
            u = stbi__convert_format(u, 4, req_comp, g->w, g->h);
    }
    else if (g->out)
        STBI_FREE(g->out);
    STBI_FREE(g);
    return u;
}

static int stbi__gif_info(stbi__context *s, int *x, int *y, int *comp)
{
    return stbi__gif_info_raw(s, x, y, comp);
}
#endif

// *************************************************************************************************
// Radiance RGBE HDR loader
// originally by Nicolas Schulz
#ifndef STBI_NO_HDR
static int stbi__hdr_test_core(stbi__context *s, const char *signature)
{
    int i;
    for (i = 0; signature[i]; ++i)
        if (stbi__get8(s) != signature[i])
            return 0;
    stbi__rewind(s);
    return 1;
}

static int stbi__hdr_test(stbi__context* s)
{
    int r = stbi__hdr_test_core(s, "#?RADIANCE\n");
    stbi__rewind(s);
    if (!r) {
        r = stbi__hdr_test_core(s, "#?RGBE\n");
        stbi__rewind(s);
    }
    return r;
}

#define STBI__HDR_BUFLEN  1024
static char *stbi__hdr_gettoken(stbi__context *z, char *buffer)
{
    int len = 0;
    char c = '\0';

    c = (char)stbi__get8(z);

    while (!stbi__at_eof(z) && c != '\n') {
        buffer[len++] = c;
        if (len == STBI__HDR_BUFLEN - 1) {
            // flush to end of line
            while (!stbi__at_eof(z) && stbi__get8(z) != '\n')
                ;
            break;
        }
        c = (char)stbi__get8(z);
    }

    buffer[len] = 0;
    return buffer;
}

static void stbi__hdr_convert(float *output, stbi_uc *input, int req_comp)
{
    if (input[3] != 0) {
        float f1;
        // Exponent
        f1 = (float)ldexp(1.0f, input[3] - (int)(128 + 8));
        if (req_comp <= 2)
            output[0] = (input[0] + input[1] + input[2]) * f1 / 3;
        else {
            output[0] = input[0] * f1;
            output[1] = input[1] * f1;
            output[2] = input[2] * f1;
        }
        if (req_comp == 2) output[1] = 1;
        if (req_comp == 4) output[3] = 1;
    }
    else {
        switch (req_comp) {
        case 4: output[3] = 1; /* fallthrough */
        case 3: output[0] = output[1] = output[2] = 0;
            break;
        case 2: output[1] = 1; /* fallthrough */
        case 1: output[0] = 0;
            break;
        }
    }
}

static float *stbi__hdr_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
  stbi__result_info *ri)
{
    char buffer[STBI__HDR_BUFLEN];
    char *token;
    int valid = 0;
    int width, height;
    stbi_uc *scanline;
    float *hdr_data;
    int len;
    unsigned char count, value;
    int i, j, k, c1, c2, z;
    const char *headerToken;
    STBI_NOTUSED(ri);

    // Check identifier
    headerToken = stbi__hdr_gettoken(s, buffer);
    if (strcmp(headerToken, "#?RADIANCE") != 0 && strcmp(headerToken, "#?RGBE") != 0)
        return stbi__errpf("not HDR", "Corrupt HDR image");

    // Parse header
    for (;;) {
        token = stbi__hdr_gettoken(s, buffer);
        if (token[0] == 0) break;
        if (strcmp(token, "FORMAT=32-bit_rle_rgbe") == 0) valid = 1;
    }

    if (!valid)    return stbi__errpf("unsupported format", "Unsupported HDR format");

    // Parse width and height
    // can't use sscanf() if we're not using stdio!
    token = stbi__hdr_gettoken(s, buffer);
    if (strncmp(token, "-Y ", 3))  return stbi__errpf("unsupported data layout", "Unsupported HDR
  format");
    token += 3;
    height = (int)strtol(token, &token, 10);
    while (*token == ' ') ++token;
    if (strncmp(token, "+X ", 3))  return stbi__errpf("unsupported data layout", "Unsupported HDR
  format");
    token += 3;
    width = (int)strtol(token, NULL, 10);

    *x = width;
    *y = height;

    if (comp) *comp = 3;
    if (req_comp == 0) req_comp = 3;

    if (!stbi__mad4sizes_valid(width, height, req_comp, sizeof(float), 0))
        return stbi__errpf("too large", "HDR image is too large");

    // Read data
    hdr_data = (float *)stbi__malloc_mad4(width, height, req_comp, sizeof(float), 0);
    if (!hdr_data)
        return stbi__errpf("outofmem", "Out of memory");

    // Load image data
    // image data is stored as some number of sca
    if (width < 8 || width >= 32768) {
        // Read flat data
        for (j = 0; j < height; ++j) {
            for (i = 0; i < width; ++i) {
                stbi_uc rgbe[4];
            main_decode_loop:
                stbi__getn(s, rgbe, 4);
                stbi__hdr_convert(hdr_data + j * width * req_comp + i * req_comp, rgbe, req_comp);
            }
        }
    }
    else {
        // Read RLE-encoded data
        scanline = NULL;

        for (j = 0; j < height; ++j) {
            c1 = stbi__get8(s);
            c2 = stbi__get8(s);
            len = stbi__get8(s);
            if (c1 != 2 || c2 != 2 || (len & 0x80)) {
                // not run-length encoded, so we have to actually use THIS data as a decoded
                // pixel (note this can't be a valid pixel--one of RGB must be >= 128)
                stbi_uc rgbe[4];
                rgbe[0] = (stbi_uc)c1;
                rgbe[1] = (stbi_uc)c2;
                rgbe[2] = (stbi_uc)len;
                rgbe[3] = (stbi_uc)stbi__get8(s);
                stbi__hdr_convert(hdr_data, rgbe, req_comp);
                i = 1;
                j = 0;
                STBI_FREE(scanline);
                goto main_decode_loop; // yes, this makes no sense
            }
            len «= 8;
            len |= stbi__get8(s);
            if (len != width) { STBI_FREE(hdr_data); STBI_FREE(scanline); return stbi__errpf("invalid decoded scanline length", "corrupt HDR"); }
            if (scanline == NULL) {
                scanline = (stbi_uc *)stbi__malloc_mad2(width, 4, 0);
                if (!scanline) {
                    STBI_FREE(hdr_data);
                    return stbi__errpf("outofmem", "Out of memory");
                }
            }

            for (k = 0; k < 4; ++k) {
                int nleft;
                i = 0;
                while ((nleft = width - i) > 0) {
                    count = stbi__get8(s);
                    if (count > 128) {
                        // Run
                        value = stbi__get8(s);
                        count -= 128;
                        if (count > nleft) { STBI_FREE(hdr_data); STBI_FREE(scanline); return stbi__errpf("corrupt", "bad RLE data in HDR"); }
                        for (z = 0; z < count; ++z)
                            scanline[i++ * 4 + k] = value;
                    }
                    else {
                        // Dump
                        if (count > nleft) { STBI_FREE(hdr_data); STBI_FREE(scanline); return stbi__errpf("corrupt", "bad RLE data in HDR"); }
                        for (z = 0; z < count; ++z)
                            scanline[i++ * 4 + k] = stbi__get8(s);
                    }
                }
            }
            for (i = 0; i < width; ++i)
                stbi__hdr_convert(hdr_data + (j*width + i)*req_comp, scanline + i * 4, req_comp);
        }
        if (scanline)
            STBI_FREE(scanline);
    }

    return hdr_data;
}

static int stbi__hdr_info(stbi__context *s, int *x, int *y, int *comp)
{
    char buffer[STBI__HDR_BUFLEN];
    char *token;
    int valid = 0;

    if (stbi__hdr_test(s) == 0) {
        stbi__rewind(s);
        return 0;
    }

    for (;;) {
        token = stbi__hdr_gettoken(s, buffer);
        if (token[0] == 0) break;
        if (strcmp(token, "FORMAT=32-bit_rle_rgbe") == 0) valid = 1;
    }

    if (!valid) {
        stbi__rewind(s);
        return 0;
    }
    token = stbi__hdr_gettoken(s, buffer);
    if (strncmp(token, "-Y ", 3)) {
        stbi__rewind(s);
        return 0;
    }
    token += 3;
    *y = (int)strtol(token, &token, 10);
    while (*token == ' ') ++token;
    if (strncmp(token, "+X ", 3)) {
        stbi__rewind(s);
        return 0;
    }
    token += 3;
    *x = (int)strtol(token, NULL, 10);
    *comp = 3;
    return 1;
}
#endif // STBI_NO_HDR

#ifndef STBI_NO_BMP
static int stbi__bmp_info(stbi__context *s, int *x, int *y, int *comp)
{
    void *p;
    stbi__bmp_data info;

    info.all_a = 255;
    p = stbi__bmp_parse_header(s, &info);
    stbi__rewind(s);
    if (p == NULL)
        return 0;
    *x = s->img_x;
    *y = s->img_y;
    *comp = info.ma ? 4 : 3;
    return 1;
}
#endif

#ifndef STBI_NO_PSD
static int stbi__psd_info(stbi__context *s, int *x, int *y, int *comp)
{
    int channelCount;
    if (stbi__get32be(s) != 0x38425053) {
        stbi__rewind(s);
        return 0;
    }
    if (stbi__get16be(s) != 1) {
        stbi__rewind(s);
        return 0;
    }
    stbi__skip(s, 6);
    channelCount = stbi__get16be(s);
    if (channelCount < 0 || channelCount > 16) {
        stbi__rewind(s);
        return 0;
    }
    *y = stbi__get32be(s);
    *x = stbi__get32be(s);
    if (stbi__get16be(s) != 8) {
        stbi__rewind(s);
        return 0;
    }
    if (stbi__get16be(s) != 3) {
        stbi__rewind(s);
        return 0;
    }
    *comp = 4;
    return 1;
}
#endif

#ifndef STBI_NO_PIC
static int stbi__pic_info(stbi__context *s, int *x, int *y, int *comp)
{
    int act_comp = 0, num_packets = 0, chained;
    stbi__pic_packet packets[10];

    if (!stbi__pic_is4(s, "\x53\x80\xF6\x34")) {
        stbi__rewind(s);
        return 0;
    }

    stbi__skip(s, 88);

    *x = stbi__get16be(s);
    *y = stbi__get16be(s);
    if (stbi__at_eof(s)) {
        stbi__rewind(s);
        return 0;
    }
    if ((*x) != 0 && (1 « 28) / (*x) < (*y)) {
        stbi__rewind(s);
        return 0;
    }

    stbi__skip(s, 8);

    do {
        stbi__pic_packet *packet;

        if (num_packets == sizeof(packets) / sizeof(packets[0]))
            return 0;

        packet = &packets[num_packets++];
        chained = stbi__get8(s);
        packet->size = stbi__get8(s);
        packet->type = stbi__get8(s);
        packet->channel = stbi__get8(s);
        act_comp |= packet->channel;

        if (stbi__at_eof(s)) {
            stbi__rewind(s);
            return 0;
        }
        if (packet->size != 8) {
            stbi__rewind(s);
            return 0;
        }
    } while (chained);

    *comp = (act_comp & 0x10 ? 4 : 3);

    return 1;
}
#endif

// *************************************************************************************************
// Portable Gray Map and Portable Pixel Map loader
// by Ken Miller
//
// PGM: http://netpbm.sourceforge.net/doc/pgm.html
// PPM: http://netpbm.sourceforge.net/doc/ppm.html
//
// Known limitations:
//    Does not support comments in the header section
//    Does not support ASCII image data (formats P2 and P3)
//    Does not support 16-bit-per-channel

#ifndef STBI_NO_PNM

static int      stbi__pnm_test(stbi__context *s)
{
    char p, t;
    p = (char)stbi__get8(s);
    t = (char)stbi__get8(s);
    if (p != 'P' || (t != '5' && t != '6')) {
        stbi__rewind(s);
        return 0;
    }
    return 1;
```

```
}

static void *stbi__pnm_load(stbi__context *s, int *x, int *y, int *comp, int req_comp,
   stbi__result_info *ri)
{
    stbi_uc *out;
    STBI_NOTUSED(ri);

    if (!stbi__pnm_info(s, (int *)&s->img_x, (int *)&s->img_y, (int *)&s->img_n))
        return 0;

    *x = s->img_x;
    *y = s->img_y;
    *comp = s->img_n;

    if (!stbi__mad3sizes_valid(s->img_n, s->img_x, s->img_y, 0))
        return stbi__errpuc("too large", "PNM too large");

    out = (stbi_uc *)stbi__malloc_mad3(s->img_n, s->img_x, s->img_y, 0);
    if (!out) return stbi__errpuc("outofmem", "Out of memory");
    stbi__getn(s, out, s->img_n * s->img_x * s->img_y);

    if (req_comp && req_comp != s->img_n) {
        out = stbi__convert_format(out, s->img_n, req_comp, s->img_x, s->img_y);
        if (out == NULL) return out; // stbi__convert_format frees input on failure
    }
    return out;
}

static int      stbi__pnm_isspace(char c)
{
    return c == ' ' || c == '\t' || c == '\n' || c == '\v' || c == '\f' || c == '\r';
}

static void     stbi__pnm_skip_whitespace(stbi__context *s, char *c)
{
    for (;;) {
        while (!stbi__at_eof(s) && stbi__pnm_isspace(*c))
            *c = (char)stbi__get8(s);

        if (stbi__at_eof(s) || *c != '#')
            break;

        while (!stbi__at_eof(s) && *c != '\n' && *c != '\r')
            *c = (char)stbi__get8(s);
    }
}

static int      stbi__pnm_isdigit(char c)
{
    return c >= '0' && c <= '9';
}

static int      stbi__pnm_getinteger(stbi__context *s, char *c)
{
    int value = 0;

    while (!stbi__at_eof(s) && stbi__pnm_isdigit(*c)) {
        value = value * 10 + (*c - '0');
        *c = (char)stbi__get8(s);
    }

    return value;
}

static int      stbi__pnm_info(stbi__context *s, int *x, int *y, int *comp)
{
    int maxv;
    char c, p, t;

    stbi__rewind(s);

    // Get identifier
    p = (char)stbi__get8(s);
    t = (char)stbi__get8(s);
    if (p != 'P' || (t != '5' && t != '6')) {
        stbi__rewind(s);
        return 0;
    }

    *comp = (t == '6') ? 3 : 1;  // '5' is 1-component .pgm; '6' is 3-component .ppm

    c = (char)stbi__get8(s);
    stbi__pnm_skip_whitespace(s, &c);

    *x = stbi__pnm_getinteger(s, &c); // read width
    stbi__pnm_skip_whitespace(s, &c);

    *y = stbi__pnm_getinteger(s, &c); // read height
    stbi__pnm_skip_whitespace(s, &c);

    maxv = stbi__pnm_getinteger(s, &c);  // read max value

    if (maxv > 255)
        return stbi__err("max value > 255", "PPM image not 8-bit");
    else
        return 1;
}
#endif

static int stbi__info_main(stbi__context *s, int *x, int *y, int *comp)
{
#ifndef STBI_NO_JPEG
    if (stbi__jpeg_info(s, x, y, comp)) return 1;
#endif

#ifndef STBI_NO_PNG
    if (stbi__png_info(s, x, y, comp))  return 1;
#endif

#ifndef STBI_NO_GIF
    if (stbi__gif_info(s, x, y, comp))  return 1;
#endif

#ifndef STBI_NO_BMP
    if (stbi__bmp_info(s, x, y, comp))  return 1;
#endif

#ifndef STBI_NO_PSD
    if (stbi__psd_info(s, x, y, comp))  return 1;
#endif

#ifndef STBI_NO_PIC
    if (stbi__pic_info(s, x, y, comp))  return 1;
#endif

#ifndef STBI_NO_PNM
    if (stbi__pnm_info(s, x, y, comp))  return 1;
#endif

#ifndef STBI_NO_HDR
    if (stbi__hdr_info(s, x, y, comp))  return 1;
#endif

    // test tga last because it's a crappy test!
#ifndef STBI_NO_TGA
    if (stbi__tga_info(s, x, y, comp))
        return 1;
#endif
    return stbi__err("unknown image type", "Image not of any known type, or corrupt");
}

#ifndef STBI_NO_STDIO
STBIDEF int stbi_info(char const *filename, int *x, int *y, int *comp)
{
    FILE *f = stbi__fopen(filename, "rb");
    int result;
    if (!f) return stbi__err("can't fopen", "Unable to open file");
    result = stbi_info_from_file(f, x, y, comp);
    fclose(f);
    return result;
}

STBIDEF int stbi_info_from_file(FILE *f, int *x, int *y, int *comp)
{
    int r;
    stbi__context s;
    long pos = ftell(f);
    stbi__start_file(&s, f);
    r = stbi__info_main(&s, x, y, comp);
    fseek(f, pos, SEEK_SET);
    return r;
}
#endif // !STBI_NO_STDIO

STBIDEF int stbi_info_from_memory(stbi_uc const *buffer, int len, int *x, int *y, int *comp)
{
    stbi__context s;
    stbi__start_mem(&s, buffer, len);
    return stbi__info_main(&s, x, y, comp);
}

STBIDEF int stbi_info_from_callbacks(stbi_io_callbacks const *c, void *user, int *x, int *y, int *comp)
{
    stbi__context s;
    stbi__start_callbacks(&s, (stbi_io_callbacks *)c, user);
    return stbi__info_main(&s, x, y, comp);
}

#endif // STB_IMAGE_IMPLEMENTATION

/*
revision history:
2.13  (2016-11-29) add 16-bit API, only supported for PNG right now
2.12  (2016-04-02) fix typo in 2.11 PSD fix that caused crashes
2.11  (2016-04-02) allocate large structures on the stack
remove white matting for transparent PSD
fix reported channel count for PNG & BMP
re-enable SSE2 in non-gcc 64-bit
support RGB-formatted JPEG
read 16-bit PNGs (only as 8-bit)
2.10  (2016-01-22) avoid warning introduced in 2.09 by STBI_REALLOC_SIZED
2.09  (2016-01-16) allow comments in PNM files
16-bit-per-pixel TGA (not bit-per-component)
info() for TGA could break due to .hdr handling
info() for BMP to shares code instead of sloppy parse
can use STBI_REALLOC_SIZED if allocator doesn't support realloc
code cleanup
2.08  (2015-09-13) fix to 2.07 cleanup, reading RGB PSD as RGBA
2.07  (2015-09-13) fix compiler warnings
partial animated GIF support
limited 16-bpc PSD support
#ifdef unused functions
bug with < 92 byte PIC,PNM,HDR,TGA
2.06  (2015-04-19) fix bug where PSD returns wrong '*comp' value
2.05  (2015-04-19) fix bug in progressive JPEG handling, fix warning
2.04  (2015-04-15) try to re-enable SIMD on MinGW 64-bit
2.03  (2015-04-12) extra corruption checking (mmozeiko)
stbi_set_flip_vertically_on_load (nguillemot)
fix NEON support; fix mingw support
2.02  (2015-01-19) fix incorrect assert, fix warning
2.01  (2015-01-17) fix various warnings; suppress SIMD on gcc 32-bit without -msse2
2.00b (2014-12-25) fix STBI_MALLOC in progressive JPEG
2.00  (2014-12-25) optimize JPG, including x86 SSE2 & NEON SIMD (ryg)
progressive JPEG (stb)
PGM/PPM support (Ken Miller)
STBI_MALLOC,STBI_REALLOC,STBI_FREE
GIF bugfix -- seemingly never worked
STBI_NO_*, STBI_ONLY_*
1.48  (2014-12-14) fix incorrectly-named assert()
1.47  (2014-12-14) 1/2/4-bit PNG support, both direct and paletted (Omar Cornut & stb)
optimize PNG (ryg)
fix bug in interlaced PNG with user-specified channel count (stb)
1.46  (2014-08-26)
fix broken tRNS chunk (colorkey-style transparency) in non-paletted PNG
1.45  (2014-08-16)
fix MSVC-ARM internal compiler error by wrapping malloc
1.44  (2014-08-07)
various warning fixes from Ronny Chevalier
1.43  (2014-07-15)
fix MSVC-only compiler problem in code changed in 1.42
1.42  (2014-07-09)
don't define _CRT_SECURE_NO_WARNINGS (affects user code)
fixes to stbi__cleanup_jpeg path
added STBI_ASSERT to avoid requiring assert.h
1.41  (2014-06-25)
fix search&replace from 1.36 that messed up comments/error messages
1.40  (2014-06-22)
fix gcc struct-initialization warning
1.39  (2014-06-15)
fix to TGA optimization when req_comp != number of components in TGA;
fix to GIF loading because BMP wasn't rewinding (whoops, no GIFs in my test suite)
add support for BMP version 5 (more ignored fields)
1.38  (2014-06-06)
suppress MSVC warnings on integer casts truncating values
fix accidental rename of 'skip' field of I/O
1.37  (2014-06-04)
remove duplicate typedef
1.36  (2014-06-03)
convert to header file single-file library
if de-iphone isn't set, load iphone images color-swapped instead of returning NULL
1.35  (2014-05-27)
various warnings
fix broken STBI_SIMD path
fix bug where stbi_load_from_file no longer left file pointer in correct place
fix broken non-easy path for 32-bit BMP (possibly never used)
TGA optimization by Arseny Kapoulkine
1.34  (unknown)
use STBI_NOTUSED in stbi__resample_row_generic(), fix one more leak in tga failure case
1.33  (2011-07-14)
make stbi_is_hdr work in STBI_NO_HDR (as specified), minor compiler-friendly improvements
```

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

4.9 Texture.h

```
#pragma once
// GLEW
#include <GL/glew.h>

// Other Libs
#include "stb_image.h"

// Other includes
#include "Model.h"
#include <vector>


class TextureLoading
{
public:
    static GLuint LoadTexture(GLchar *path)
    {
        unsigned int textureID;
        glGenTextures(1, &textureID);

        int width, height, nrComponents;
        unsigned char *data = stbi_load(path, &width, &height, &nrComponents, 0);
        if (data)
        {
            GLenum format;
            if (nrComponents == 1)
                format = GL_RED;
            else if (nrComponents == 3)
                format = GL_RGB;
            else if (nrComponents == 4)
                format = GL_RGBA;

            glBindTexture(GL_TEXTURE_2D, textureID);
            glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
            glGenerateMipmap(GL_TEXTURE_2D);

            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

            stbi_image_free(data);

        }
        else
        {
            std::cout « "Failed to load texture" « path « std::endl;
            stbi_image_free(data);
        }

        return textureID;
    }


    static GLuint LoadCubemap(vector<const GLchar * > faces)
    {
        GLuint textureID;
        glGenTextures(1, &textureID);

        int width, height, nrChannels;
        for (unsigned int i = 0; i < faces.size(); i++)
        {
            unsigned char *data = stbi_load(faces[i], &width, &height, &nrChannels, 0);
            if (data)
            {
                glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, data);
                stbi_image_free(data);
            }
            else
            {
                std::cout « "Cubemap texture failed to load at path: " « faces[i] « std::endl;
                stbi_image_free(data);
            }
        }
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
        glBindTexture(GL_TEXTURE_CUBE_MAP, 0);

        return textureID;
    }

};
```

Index