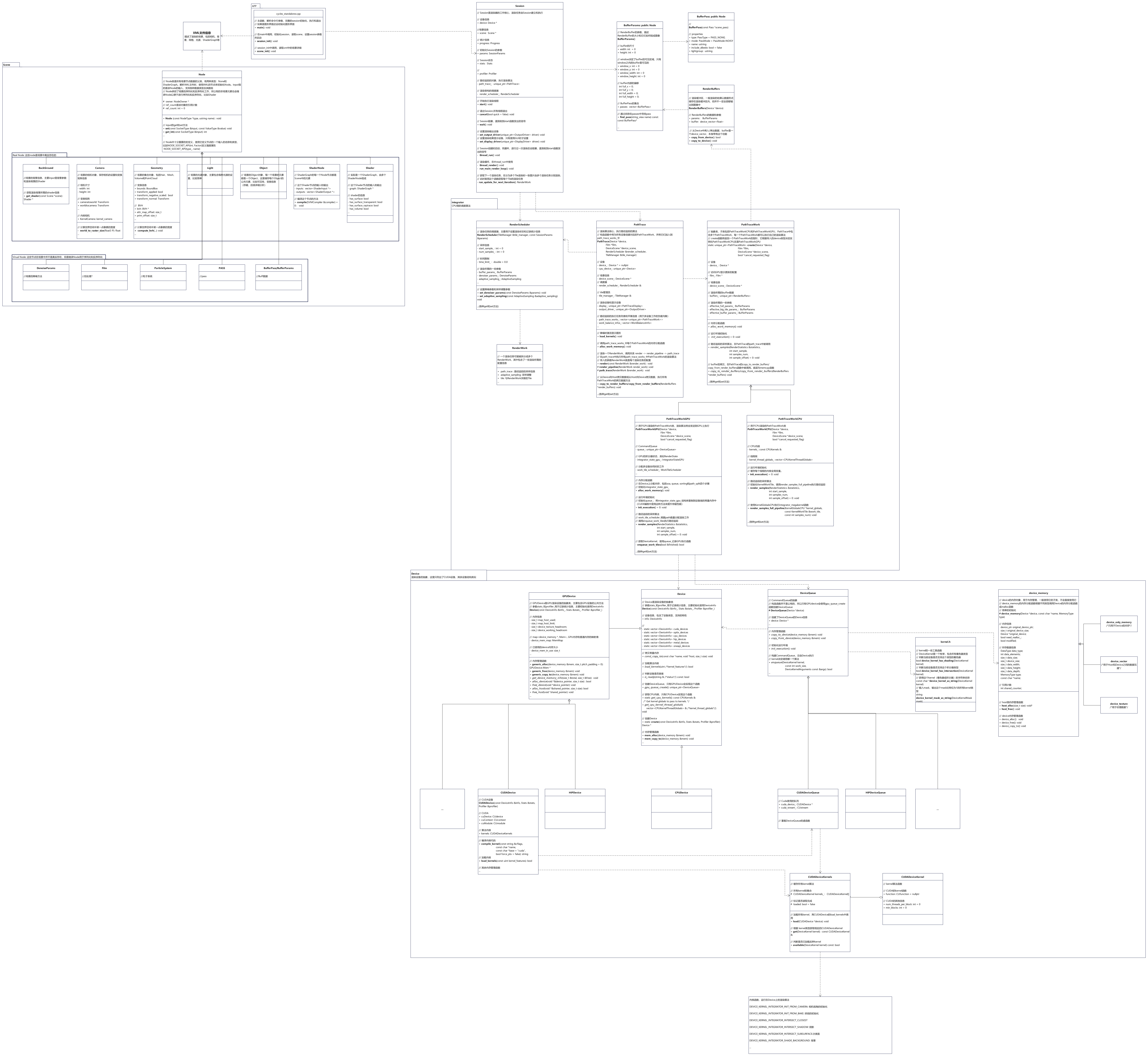

APP

cycles_standalone.cpp

// 主函数，解析命令行参数，完整的session初始化、执行和退出
// 如果是图形界面还会初始化图形界面
+ **main**(): void

// 在main中调用，初始化session，读取scene，设置session参数并启动
+ **session_init**(): void

// session_init中调用，读取xml中的场景详细
+ **scene_init**(): void

XML文件信息
描述了渲染的场景，包括相机、背景、网格、光源、ShaderGraph等

Scene

Node

// Node类是所有场景节点数据的父类，有两种类型：None和ShaderGraph。解析XML文件时，使用XML的节点来初始化Node。input指的是该Node的输入，支持各种数据类型及其数组
// Node承担了场景的序列化和反序列化工作，所以有的非场景元素也会继承Node以便于进行序列化和反序列化，比如Shader

# owner: NodeOwner *
// ref_count是该对象的引用计数
# ref_count: int = 0

+ **Node** (const NodeType *type, ustring name) : void

// input的get和set方法
+ **set**(const SocketType &input, const ValueType &value): void
+ **get_int**(const SocketType &input): int
...

// Node中十分重要的宏定义，使用它定义节点的一个输入的名称和类型，比如NODE_SOCKET_API(int, Factor)定义强度属性
NODE_SOCKET_API(type_, name)

Real Node: 这些node是场景中真实存在的

BackGround

//场景的背景信息，主要input是背景参数和渲染背景的Shader

// 获取渲染背景所需的shader信息
+ **get_shader**(const Scene *scene): Shader *

Camera

// 场景的相机对象，保存相机的设置和变换矩阵信息

// 相机尺寸
- width: int
- height: int

// 变换矩阵
+ cameratoworld: Transform
+ worldtocamera: Transform
...

// 内核相机
+ KernelCamera: kernel_camera

// 计算世界空间中某一点像素的宽度
- **world_to_raster_size**(float3 P): float
...

Geometry

// 场景的集合对象，包括Hair、Mesh、Volume和PointCloud

// 变换信息
+ bounds: BoundBox
+ transform_applied: bool
+ transform_negative_scaled: bool
+ transform_normal: Transform

// BVH
+ bvh: BVH *
+ attr_map_offset: size_t
+ prim_offset: size_t
...

// 计算世界空间中某一点像素的宽度
+ **compute_bvh**(...): void
...

Light

// 场景的光源对象，主要包含每种光源的设置，比较简单

Object

// 场景的Object对象，每一个场景的元素都是一个Object，这里储存每个Object的公共元素，比如可见性，变换信息（存疑，后续详细分析）

ShaderNode

// ShaderGraph的每一个Node节点都是Scene中的元素

// 这个Shader节点的输入和输出
- inputs: vector<ShaderInput *>
- outputs: vector<ShaderOutput *>

// 编译这个节点的方法
+ **compile**(SVMCompiler &compiler) = 0: void

Shader

// 实际是一个ShaderGraph，由多个ShaderNode组成

// 这个Shader节点的输入和输出
- graph: ShaderGraph *

// shader的信息
- has_surface: bool
- has_surface_transparent: bool
- has_surface_raytrace: bool
- has_volume: bool
...

Virual Node: 这些节点在场景中并不是真实存在，但是继承Node用于序列化和反序列化

DenoiseParams

//场景的降噪方法

Film

//后处理?

ParticleSystem

//粒子系统

PASS

//pass

BufferPass/BufferParams

//Buff数据

Session

// Session是渲染器的工作核心，渲染任务由Session建立和执行

// 设备信息
+ device: Device *

//场景信息
+ scene: Scene *

// 统计信息
+ progress: Progress

// 初始化Session的参数
+ params: SessionParams

// Session状态
+ stats: Stats

//
+ profiler: Profiler

// 路径追踪的对象，执行渲染算法
- path_trace_: unique_ptr<PathTrace>

// 渲染架构的调度器
- render_scheduler_: RenderScheduler

// 开始执行渲染线程
+ **start**(): void

// 退出Session,所有线程退出
+ **cancel**(bool quick = false): void

// Session阻塞，直到收到start函数发出的信号
+ **wait**(): void

// 设置渲染输出设备
+ **set_output_driver**(unique_ptr<OutputDriver> driver): void
// 设置渲染结果显示设备，只有使用GUI时才设置
+ **set_display_driver**(unique_ptr<DisplayDriver> driver): void

// Session创建时启动，死循环，进行过一次渲染后会阻塞，直到收到start函数发出的信号
- **thread_run**(): void

// 渲染循环，在thread_run中使用
- **thread_render**(): void
- **run_main_render_loop**(): void

// 获取下一个渲染任务，在分为多个Tile渲染时一张图片由多个渲染任务分别渲染，
// 此时就用这个函数获取每个Tile的渲染任务
- **run_update_for_next_iteration**(): RenderWork

BufferParams: public Node

// RenderBuffer的参数，描述RenderBuffer的大小和它们如何组成图像
BufferParams()

// buffer的尺寸
+ width: int = 0
+ height: int = 0

// window决定了buffer的可见区域，只有window之内的buffer是可见的
+ window_x: int = 0
+ window_y: int = 0
+ window_width: int = 0
+ window_height: int = 0

// buffer内部的偏移
int full_x = 0;
int full_y = 0;
int full_width = 0;
int full_height = 0;

// BufferPass的集合
+ passes: vector<BufferPass>

// 通过名称在passes中寻找pass
+ **find_pass**(string_view name) const: const BufferPass*

...

BufferPass: public Node

//
BufferPass(const Pass *scene_pass)

// properties
+ type: PassType = PASS_NONE;
+ mode: PassMode = PassMode::NOISY
+ name: ustring
+ include_albedo: bool = false
+ lightgroup: ustring

RenderBuffers

// 渲染缓冲区，一般渲染的结果以数据形式储存在渲染缓冲区内，但并不一定全部都输出到图像中
RenderBuffers(Device *device)

// RenderBuffers的数据和参数
+ params: BufferParams
+ buffer: device_vector<float>

// 从Device中拷入/拷出数据，buffer是一个device_vector，本身带有这个功能
+ **copy_from_device**(): bool
+ **copy_to_device**(): void

integrator
CPU侧的调度算法

RenderScheduler

// 渲染任务的调度器，主要用于设置渲染状态和记录统计信息
RenderScheduler(TileManager &tile_manager, const SessionParams &params)

// 采样信息
- start_sample_ : int = 0
- num_samples_ : int = 0

// 时间限制
- time_limit_ : double = 0.0

// 渲染所需的一些参数
- buffer_params_: BufferParams
- denoiser_params_: DenoiseParams
- adaptive_sampling_: AdaptiveSampling
...

// 设置降噪参数和采样调整参数
+ **set_denoiser_params**(const DenoiseParams &params): void
+ **set_adaptive_sampling**(const AdaptiveSampling &adaptive_sampling): void

...(各种get和set方法)

RenderWork

// 一个渲染任务可能被拆分成多个RenderWork，其中包含了一些渲染所需的配置信息

+ path_trace : 路径追踪的采样信息
+ adaptive_sampling: 采样调整
+ tile: 与RenderWork关联的Tile
...

PathTrace

// 渲染算法核心，执行路径追踪的算法
// 构造函数中将为所有设备创建对应的PathTraceWork，并将它们加入到path_trace_works_中
PathTrace(Device *device,
Film *film,
DeviceScene *device_scene,
RenderScheduler &render_scheduler,
TileManager &tile_manager)

// 设备
- device_ : Device * = nullptr
- cpu_device_: unique_ptr<Device>

// 场景信息
- device_scene_: DeviceScene *
// 调度器
- render_scheduler_: RenderScheduler &

// tile管理员
- tile_manager_: TileManager &

// 渲染设备和显示设备
- display_: unique_ptr<PathTraceDisplay>
- output_driver_: unique_ptr<OutputDriver>

// 路径追踪的执行任务列表和平衡信息（用于多设备工作的负载均衡）
- path_trace_works_: vector<unique_ptr<PathTraceWork>>
- work_balance_infos_: vector<WorkBalanceInfo>
...

// 降噪时激活显示图形
+ **load_kernels**(): void

// 调用path_trace_works_中每个PathTraceWork的内存分配函数
+ **alloc_work_memory**(): void

// 渲染一个RenderWork，调用关系 render -> render_pipeline -> path_trace
// 在path_trace中执行所有path_trace_works_中PathTraceWork的渲染算法
// 传入的参数RenderWork就是每个渲染任务的配置
+ **render**(const RenderWork &render_work) : void
# **render_pipeline**(RenderWork render_work): void
# **path_trace**(RenderWork &render_work): void

// 从Device向Host拷贝数据或从Host向Device拷贝数据，执行所有PathTraceWork的拷贝数据方法
+ **copy_to_render_buffers/copy_from_render_buffers**(RenderBuffers *render_buffers): void

...(各种get和set方法)

PathTraceWork

// 抽象类，子类包括PathTraceWorkCPU和PathTraceWorkGPU，PathTrace中包含多个PathTraceWork，每一个PathTraceWork都可以执行自己的渲染算法
// create函数将返回一个PathTraceWork的指针，它根据传入的device类型决定实例化PathTraceWorkCPU还是PathTraceWorkGPU
static unique_ptr<PathTraceWork> **create**(Device *device,
Film *film,
DeviceScene *device_scene,
bool *cancel_requested_flag)

// 设备
- device_ : Device *

// 访问GPU显示更新的配置
- film_: Film *

// 场景信息
- device_scene_: DeviceScene *

// 渲染所需的buffer数据
- buffers_: unique_ptr<RenderBuffers>

// 渲染所需的一些参数
- effective_full_params_: BufferParams
- effective_big_tile_params_: BufferParams
- effective_buffer_params_: BufferParams
...

// 内存分配函数
+ *alloc_work_memory*(): void

// 运行环境初始化
+ *init_execution*() = 0: void

// 路径追踪的采样算法，在PathTrace的path_trace中被调用
+ *render_samples*(RenderStatistics &statistics,
int start_sample,
int samples_num,
int sample_offset) = 0: void

// buffer的拷贝，在PathTrace的copy_to_render_buffers/copy_from_render_buffers函数中被调用。底层为memcpy函数
+ *copy_to_render_buffers/copy_from_render_buffers*(RenderBuffers *render_buffers): void

...(各种get和set方法)

PathTraceWorkGPU

// 用于GPU渲染的PathTraceWork类，渲染算法将会发送到GPU上执行
PathTraceWorkGPU(Device *device,
Film *film,
DeviceScene *device_scene,
bool *cancel_requested_flag)

// CommandQueue
- queue_: unique_ptr<DeviceQueue>

// GPU的积分器状态，类似RenderState
- integrator_state_gpu_: IntegratorStateGPU

// 分配多设备协同时的工作
- work_tile_scheduler_: WorkTileScheduler
...

// 内存分配函数
// 在Device上分配内存，包括soa, queue, sorting和path_split四个步骤
// 初始化integrator_state_gpu_
+ **alloc_work_memory**(): void

// 运行环境初始化
// 初始化queue_；将integrator_state_gpu_结构体复制到设备端的常量内存中（CUDE编程中使用这种方法来提升传输性能）
+ **init_execution**() = 0: void

// 路径追踪的采样算法
// work_tile_scheduler_根据path数量分配渲染工作
// 调用enqueue_work_tiles执行路径追踪
+ **render_samples**(RenderStatistics &statistics,
int start_sample,
int samples_num,
int sample_offset) = 0: void

// 获取DeviceKernel，使用queue_记录GPU执行函数
- **enqueue_work_tiles**(bool &finished): bool

...(各种get和set方法)

PathTraceWorkCPU

// 用于CPU渲染的PathTraceWork类
PathTraceWorkCPU(Device *device,
Film *film,
DeviceScene *device_scene,
bool *cancel_requested_flag)

// CPU内核
- kernels_: const CPUKernels &

// 线程核
- kernel_thread_globals_: vector<CPUKernelThreadGlobals>

// 运行环境初始化
// 缓存每个线程的内核全局变量。
+ **init_execution**() = 0: void

// 路径追踪的采样算法
// 初始化KernelWorkTile，调用render_samples_full_pipeline执行路径追踪
+ **render_samples**(RenderStatistics &statistics,
int start_sample,
int samples_num,
int sample_offset) = 0: void

// 使用KernelGlobalsCPU执行integrator_megakernel函数
+ **render_samples_full_pipeline**(KernelGlobalsCPU *kernel_globals,
const KernelWorkTile &work_tile,
const int samples_num): void

...(各种get和set方法)

Device
渲染设备的抽象，这里只列出了CUDA设备，其余设备结构类似

GPUDevice

// GPUDevice是GPU渲染设备的抽象类，主要包含GPU设备的公共方法
// 参数stats_和profiler_用于记录统计信息，主要初始化使用DeviceInfo
Device(const DeviceInfo &info_, Stats &stats_, Profiler &profiler_)

// 内存信息
- size_t map_host_used;
- size_t map_host_limit;
- size_t device_texture_headroom;
- size_t device_working_headroom

// map<device_memory *, Mem>, GPU内存和普通内存的映射表
- device_mem_map: MemMap

// 已使用的device内存大小
- device_mem_in_use: size_t
...

// 内存管理函数
+ **generic_alloc**(device_memory &mem, size_t pitch_padding = 0): GPUDevice::Mem *
+ **generic_free**(device_memory &mem): void
+ **generic_copy_to**(device_memory &mem): void
+ *get_device_memory_info*(size_t &total, size_t &free): void
+ *alloc_device*(void *&device_pointer, size_t size) : bool
+ *free_device*(void *device_pointer): void
+ *alloc_host*(void *&shared_pointer, size_t size): bool
+ *free_host*(void *shared_pointer): void
...

Device

// Device是渲染设备的抽象类
// 参数stats_和profiler_用于记录统计信息，主要初始化使用DeviceInfo
Device(const DeviceInfo &info_, Stats &stats_, Profiler &profiler_)

// 设备信息，包含了设备类型，支持的特性
+ info: DeviceInfo

//
- static vector<DeviceInfo> cuda_devices
- static vector<DeviceInfo> optix_devices
- static vector<DeviceInfo> cpu_devices
- static vector<DeviceInfo> hip_devices
- static vector<DeviceInfo> metal_devices
- static vector<DeviceInfo> oneapi_devices

// 拷贝常量内存
+ *const_copy_to*(const char *name, void *host, size_t size): void

// 加载算法内核
+ *load_kernels*(uint /*kernel_features*/): bool

// 判断设备是否就绪
+ *is_ready*(string & /*status*/) const: bool

// 创建DeviceQueue，只有GPUDevice会实现这个函数
+ *gpu_queue_create*(): unique_ptr<DeviceQueue>

// 获取CPU内核，只有CPUDevice实现这个函数
+ static *get_cpu_kernels*(): const CPUKernels &
/* Get kernel globals to pass to kernels. */
+ *get_cpu_kernel_thread_globals*(
vector<CPUKernelThreadGlobals> & /*kernel_thread_globals*/): void

// 创建Device
+ static **create**(const DeviceInfo &info, Stats &stats, Profiler &profiler): Device *

// 内存管理函数
+ **mem_alloc**(device_memory &mem): void
+ **mem_copy_to**(device_memory &mem): void
...

DeviceQueue

// CommandQueue的抽象
// 构造函数并不是公有的，所以只有GPUdevice会使用gpu_queue_create函数创建DeviceQueue
DeviceQueue(Device *device)

// 创建了DeviceQueue的Device信息
+ device: Device *
...

// 内存管理函数
+ *copy_to_device*(device_memory &mem): void
+ *copy_from_device*(device_memory &mem): void

// 初始化运行环境
+ *init_execution*(): void

// 构建CommandQueue，交由Device执行
// kernel决定调用哪一个算法
+ *enqueue*(DeviceKernel kernel,
const int work_size,
DeviceKernelArguments const &args): bool
...

kernel.h

// kernel的一些工具函数
// DeviceKernel是一个枚举，包含所有着色器类型
// 判断当前设备是否支持这个类型的着色器
bool **device_kernel_has_shading**(DeviceKernel kernel)
// 判断当前设备是否支持这个积分器类型
bool **device_kernel_has_intersection**(DeviceKernel kernel)
// 获得这个kernel（着色器或积分器）的字符串名称
const char ***device_kernel_as_string**(DeviceKernel kernel)
// 输入mask，输出这个mask比特位为1的所有kernel类型
string
device_kernel_mask_as_string(DeviceKernelMask mask)

device_memory

// device的内存对象，用于内存管理，一般使用它的子类，不会直接使用它
// device_memory的内存分配函数根据不同类型调用Device的内存分配函数或malloc函数
// 简单的初始化
# **device_memory**(Device *device, const char *name, MemoryType type)

// 内存信息
device_ptr original_device_ptr;
size_t original_device_size;
Device *original_device;
bool need_realloc_;
bool modified;

// 所存数据信息
DataType data_type;
int data_elements;
size_t data_size;
size_t device_size;
size_t data_width;
size_t data_height;
size_t data_depth;
MemoryType type;
const char *name;

// 引用计数
int shared_counter;

// host侧内存管理函数
+ **host_alloc**(size_t size): void*
+ **host_free**(): void

// device内存管理函数
+ device_alloc(): void
+ device_free(): void
+ device_copy_to(): void

device_only_memory
/*只用于Device的内存*/

device_vector
/*用于Host和Device之间的数据互通*/

device_texture
/*用于纹理数据*/

...

CUDADevice

// CUDA设备
CUDADevice(const DeviceInfo &info, Stats &stats, Profiler &profiler)

// CUDA
+ cuDevice: CUdevice
+ cuContext: CUcontext
+ cuModule: CUmodule

// 算法内核
+ kernels: CUDADeviceKernels

// 编译内核代码
+ **compile_kernel**(const string &cflags,
const char *name,
const char *base = "cuda",
bool force_ptx = false): string
// 加载内核
+ **load_kernels**(const uint kernel_features): bool

// 其余内存管理函数
...

HIPDevice

CPUDevice

CUDADeviceQueue

// Cuda使用的队列
+ cuda_device_: CUDADevice *
+ cuda_stream_: CUstream

// 重载DeviceQueue的虚函数

HIPDeviceQueue

...

CUDADeviceKernels

// 缓存所有kernel算法

// 所有kernel的集合
# CUDADeviceKernel kernels_: CUDADeviceKernel[]

// 标记是否读取完成
# loaded: bool = false

// 加载所有kernel，再CUDADevice的load_kernels中调用
+ **load**(CUDADevice *device): void

// 根据 kernel类型获取相应的CUDADeviceKernel
+ **get**(DeviceKernel kernel): const CUDADeviceKernel &

// 判断是否已加载此种kernel
+ **available**(DeviceKernel kernel) const: bool

CUDADeviceKernel

// kernel算法函数

// CUDA的kernel函数
+ function: CUfunction = nullptr

// CUDA的其他信息
+ num_threads_per_block: int = 0
+ min_blocks: int = 0

内核函数，运行在Device上的渲染算法

DEVICE_KERNEL_INTEGRATOR_INIT_FROM_CAMERA: 相机视角的初始化

DEVICE_KERNEL_INTEGRATOR_INIT_FROM_BAKE: 烘焙的初始化

DEVICE_KERNEL_INTEGRATOR_INTERSECT_CLOSEST

DEVICE_KERNEL_INTEGRATOR_INTERSECT_SHADOW: 阴影

DEVICE_KERNEL_INTEGRATOR_INTERSECT_SUBSURFACE:次表面

DEVICE_KERNEL_INTEGRATOR_SHADE_BACKGROUND: 背景

...