# Jornam Engine - Documentation ver. 0.1

Capita Selecta 2019

Joram Wessels - 6627404
Utrecht University

July 10, 2019

## 1 Overview

The *Jornam Engine* is a work in progress (the project name comes from my gamer tag). It is based on *SDL*, for the communication with the OS, and *NVidia Optix Prime++*, for the ray-triangle intersections. After initializing the SDL framework and the `Game` object, the main loop is as follows:

- Handle user input (SDL)
- Game tick
- Copy intermediate screen buffer to SDL buffer

The game object creates the `Renderer`, `Camera`, and `Scene` objects. At initialization, it passes the scene and the intermediate screen buffer to the renderer. The `render()` function is called at the end of every game tick, which takes a camera object as input and fills the intermediate screen buffer during rendering. For now, the engine provides a ray tracer without shadow rays, diffuse Phong shading, ambient light, WASD- and mouse controls, scene loading, wavefront meshes, and jpeg- and png texture mapping. It runs at a stable 57 FPS @ 720p on a test rig with a GTX 1070.

### 1.1 Classes

#### 1.1.1 Logger

The `Logger` handles exception throwing, printing, and logging to a file. It uses three separate severity thresholds for each case, which can be set depending on the stage of production. One static logger instance is used for the entire application.

### 1.1.2 Surface

The `Surface` class is taken from the assignment template from Advance Graphics , and has been stripped to the bare necessities. It is only used as the intermediate screen buffer between the `Renderer` class and SDL. Once the screen plotting is performed straight from the GPU, this class will be obsolete.

### 1.1.3 Buffer

The `Buffer` class is taken from the Optix tutorials. It handles the allocation of ray- and hit buffers and contains the respective pointers. It is intended to be replaced by custom code.

### 1.1.4 Camera

The `Camera` class handles player movement and rotation. It also contains the virtual screen information, which it can return as a `ScreenCorners` struct.

### 1.1.5 Game

The `Game` class provides the actual implementation of the game. It initializes the `Renderer`, `Scene`, and `Camera`, and receives the user input from SDL. It calls the render function at the end of every tick by passing a `Camera` instance.

### 1.1.6 Scene

The `Scene` class keeps track of the lights, instances, meshes, textures, and skyboxes on both the host and the device. It takes a `USE_GPU` flag as input that determines the memory location of all assets. Additionally, it contains the Optix `RTPmodel` array and transformation matrix array for convenient input into the Optix context. When the application renders on the host, this class also provides functions for skybox intersections, normal interpolation, and texture interpolation. Scenes are loaded from `.scene` using a call to the `SceneParser` class.

### 1.1.7 SceneParser

The `SceneParser` class can read ASCII `.scene` files and initialize the `Scene` class. It is called from within a `Scene` object, and contains a pointer to that object to add the lights, objects, and skyboxes it parses. Whenever a scene file specifies a wavefront file or texture image, it loads them into memory by calling the `MeshMap` and `TextureMap` from the `Scene` instance respectively.

### 1.1.8 MeshMap & TextureMap

These two classes provide a simple hash map implementation for meshes and textures to prevent memory duplicates. It uses the filename as identification and returns the index of the mesh/texture in the mesh/texture arrays. When hashing a new asset, it immediately loads the contents into host- or device memory by calling the `Mesh`/`Texture` constructors.

### 1.1.9 Mesh & Texture

The `Mesh` class contains array pointers to the vertices, normals, texture coordinates, and their indices. The `Texture` class contains a pointer to a pixel buffer, a width, and a height. When both the width and height are smaller than 1, the buffer pointer is instead used as a color value for solid color textures. Both of these classes load the contents into the memory specified by the `onDevice` flag during their construction.

### 1.1.10 Object3D

The `Object3D` class represents an instance of a 3D object. It contains the mesh- and texture indices, the transform, the Phong material variables, and the Optix Prime handle. It adds the triangles to the Optix context on creation.

### 1.1.11 Renderer & OptixRenderer

All renderers inherit from the `Renderer` class, which contains the most basic aspects of any renderer (screen buffer, width, height, scene, and a function that draws the world axes). The `OptixRenderer` class is one of these, which itself is another virtual superclass for renderers using Optix. The only thing it adds is the initialization of the `RTPcontext`.

### 1.1.12 RayTracer

The main renderer of this version is defined in the `RayTracer` class, which inherits from `OptixRenderer`. In addition to the inherited variables, it contains the ray- and hit buffers, as well as a pointer to the screen buffer on the GPU. The `render()` function creates the primary rays, traces them using Optix, and turns the resulting hits into pixels.

### 1.1.13 RayKernels.cu

If the application runs on the device, the `RayTracer` class calls the two CUDA kernels: `createPrimaryRaysOnDevice` and `shadeHitsOnDevice`. These functions have the same implementation as their CPU counterparts. Additionally, this CUDA file also provides the GPU versions of the normal- and texture interpolation, previously found in the `Scene` class.

## 2 Architecture

The initial idea of the project was to make a path tracer that featured advanced filtering methods. I would find my way there by iteratively making more complex renderers, which would all feature the same interface and could therefore be used as a comparison to one another using the `SideBySideRenderer`. This interface was based on the virtual `Renderer` class and its three virtual functions:

- `init()`, called once at the start of the application, using a scene instance and a screen buffer as parameters.
- `tick()`, called at the start of every game tick without any parameters, with the intention of clearing the buffers etc.
- `render()`, called at the end of every game tick with a camera instance as parameter, which would fill the screen buffer with pixel values.

Within the `render()` function, the `traceRays()` function would take an array of primary rays and the scene, and return an array of collisions, which included distance, surface normal, and color.

Due to the way instances are given to the Optix context, the `Scene` class needed to be adapted. It now contains an `RTPmodel` array and a reference to the `RTPcontext`. However, that makes the approach a lot less modular, since another renderer without Optix would still need the Optix SDK to compile the `Scene` class. Furthermore, the hit structs Optix returns only contain the instance Idx, triangle Idx, u- and v coordinate.

These output values did show me how to reorganize my architecture. I now load 4 arrays (meshes, textures, lights, and instances) to the GPU. The Optix instance index corresponds to the instances in the `Object3D` array. The `Object3D` contains the indices of the corresponding mesh and texture. The `Mesh` contains the indices of the corresponding normal vectors, which matches the triangle index of the Optix hit. Resolving those indices in the array of normal vectors results in the three normals, which can be interpolated using the(u, v) coordinates also outputted by Optix. This allows for more flexibility in handling normals than the original architecture, e.g. normal mapping.