Tutorial 1
OSPRay – Rendering code

Iluminação e Visualização II

Luís Paulo Santos, Abril 2021

Throughout this tutorial you will overview the code structure of OSPRay's renderers. We will also develop some very fundamental renderers.

Modules

OSPRay's functionality can be extended via runtime loadable plugins, referred to as modules, which are implemented as shared libraries. These modules can implement new components for OSPRay, such as renderers, geometric primitives, materials, cameras, frame buffers, etc. Dynamically loadable are thus a way of extending OSPRay without having to change its core code. Throughout this course we will concentrate on developing new renderers.

Renderer class

Below you can find a textual description of the general relationships between rendering related classes and methods. The figure below depicts this diagrammatically.

Universidade do Minho

OSPRay defines the `Renderer` class, from where other classes implementing renderers should inherit (you can see `Renderer.cpp` and `Renderer.h` on your OSPRAy's source tree, if you have the sources installed in your computer). The main methods within `Renderer` are:

- `Renderer()` – constructor;
- `~Renderer()` – destructor;
- `virtual std::string toString()` - used for debugging and error reporting purposes;
- `virtual void commit()`- used for setting the renderer parameters;
- `virtual void *beginFrame(FrameBuffer *fb)` – called by OSPRay before rendering a frame; can return a pointer to per frame data;
- `virtual float renderFrame(FrameBuffer *fb, const uint32 fbChannelFlags)` - called by OSPRay to render a frame; the default implementation in `Renderer.cpp` calls the `renderFrame()` method in the `LoadBalancer` class, which will in turn call the `Renderer`'s `renderTile()` method;
- `virtual void renderTile(void *perFrameData, Tile &tile, size_t jobID)` – used to render a tile of the frame;
- `virtual void endFrame(void *perFrameData, const int32 fbChannelFlags)` – called once at the end of the frame rendering process.

An important point is that OSPRay is designed to be a highly efficient renderer. Therefore, most of its rendering code isn't actually written in C++, but rather on a variant of C used by Intel SPMD Program Compiler (`ispc`).

You have to keep in mind that many C++ methods in the `.cpp` files call methods written in ispc (`.ispc` files). These ispc methods are appropriate for being executed using the processor highly parallel vectorial processing units (that is, the Intel processors AVX units). This is very important for operations that are applied to multiple data instances, since vectorial instruction will be used. Examples include applying the same operation to many pixels or doing the same computation to many rays (e.g., tracing).

In practice this means that some methods in the `.cpp` files call the corresponding method in the `.ispc` file and the respective processing code is actually written in ispc. The following text and the image below explain which ispc methods are called by each C++ method, for the rendering case.

If you can access OSPRay sources you can open the files `Renderer.ispc` and `Renderer.ih`. There you will find the following methods:

- `export void Renderer_set(…)`

- `void Renderer_Constructor(…)`
- `export void *uniform Renderer_beginFrame(void *uniform _self, void *uniform _fb)`
- `export void Renderer_renderTile(void *uniform _self, void *uniform perFrameData, uniform Tile &tile, uniform int jobID)`
- `void Renderer_default_renderSample(uniform Renderer *uniform self, void *uniform perFrameData, varying ScreenSample &sample)`
- `export void Renderer_endFrame(void *uniform _self, void *uniform perFrameData)`

The methods preceded by the `export` keyword are visible to the C++ Renderer class described above. Even though a sophisticated renderer can organize itself in a quite different manner, the general code organization is as follows:

- `Renderer.commit()` calls `Renderer_set()` in order to set ispc context renderer parameters;
- `Renderer.beginFrame()` calls `Renderer_beginFrame()`;
- `Renderer.renderTile()` calls `Renderer_renderTile()` in order to render a tile of pixels. `Renderer_renderTile()` is usually organized such that it calls a renderSample method for each sample of the image plane. For the default renderer this is `Renderer_default_renderSample()`.
- `Renderer.endFrame()` calls `Renderer_endFrame()`;

Above is an overall description of the default renderer organization. When defining a module two additional methods are required. On the `ispc` context an exported method for creating the renderer is required. This should have the following signature (although its name can be any):

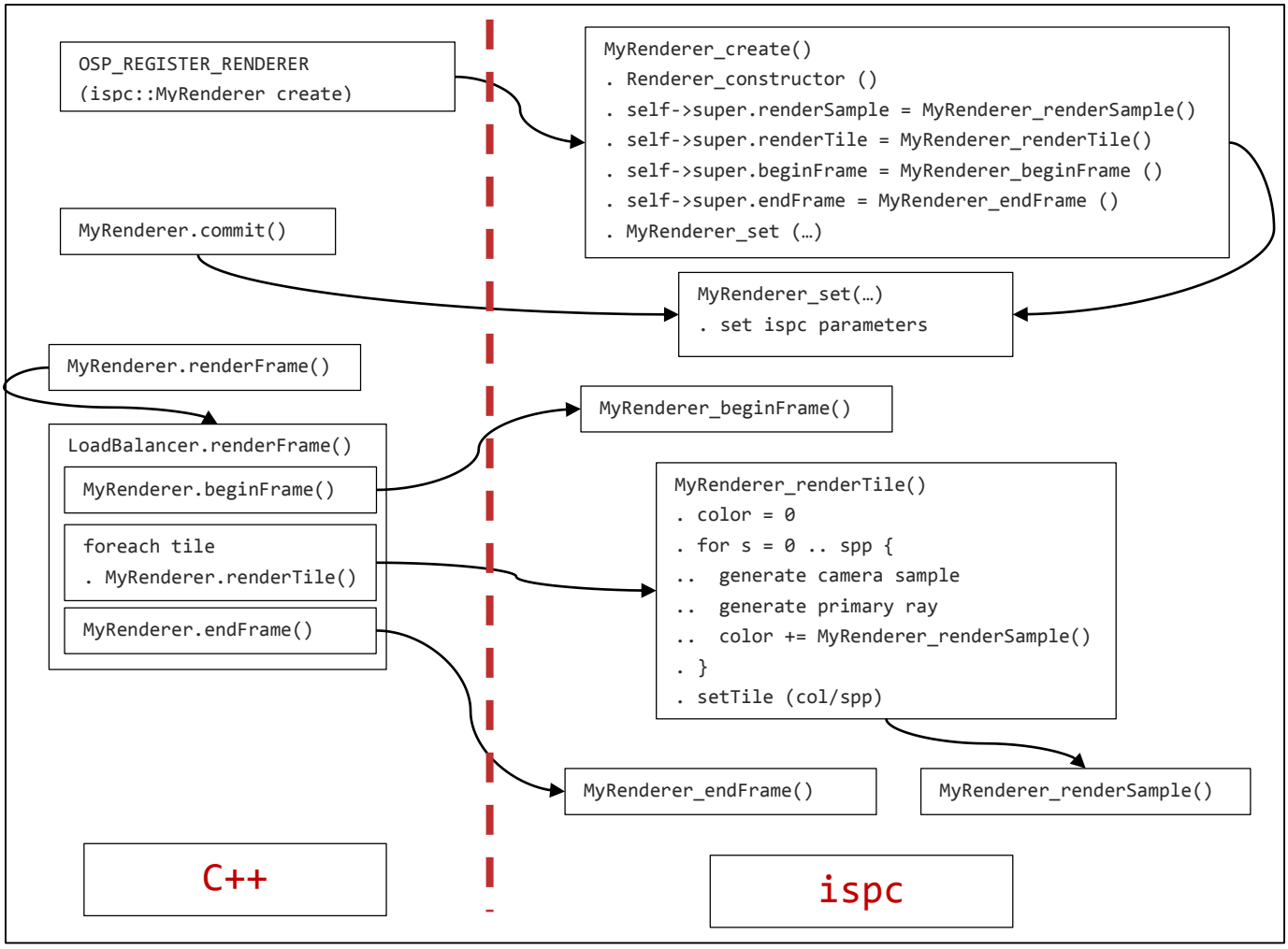
```
export void *uniform Renderer_create_renderDefault(void *uniform cppE);
```

This is the method that will call `Renderer_Constructor()` upon creation of the renderer.

Additionally, within the C++ context a method is required to be called when the module is loaded. This method does not play any relevant role, but it must exist. If the module is named `ThisModule` then this method must have the following signature (and exactly the name as it has been built below):

```
extern "C" OSPRAY_DLLEXPORT void ospray_init_module_ThisModule()
```

In many publicly available OSPRay modules this method is found on a `.cpp` file named `moduleInit.cpp` and we will maintain the same practice.



MyRenderer module

Throughout these tutorials we will be working with a renderer module referred to as `MyRenderer`. If you installed OSPray from the git repository (as indicated in Tutorial 0) then you can find the code for `MyRenderer` in the folder `ospray-vi2/src/rtlibrary/render`. This folder contains the following files:

`MyRenderer.cpp` `MyRenderer.h` `Myrenderer.ispc` `Myrenderer.iH`

>>> MyRenderer.h

This file contains the declaration of the `MyRenderer` class, which inherits from `Renderer`:

```
struct MyRenderer : public Renderer
{
```

```
        MyRenderer();
        virtual std::string toString() const override;
};
```

>>> MyRenderer.cpp

This file contains the definition of the `MyRenderer` class. It includes `"MyRenderer_ispc.h"` that is generated automatically by the `ispc` compiler, when it compiles the `MyRenderer.ispc` file. Therefore, this file will not exist the first time you build your project. Afterwards, you will find it in `ospray-vi2/build/src/rtlibrary`.

```
#include "MyRenderer.h"
// 'export'ed functions from the ispc file:
#include "MyRenderer_ispc.h"
```

The constructor explicitly builds the corresponding data structures in `ispc` context:

```
MyRenderer::MyRenderer()
{
        ispcEquivalent = ispc::MyRenderer_create(this);
}
```

The `toString` method inherited from `Renderer` must be overridden:

```
std::string MyRenderer::toString() const
{
        return "ospray::MyRenderer";
}
```

Note that none of the main rendering methods of the super class `Renderer` were overridden and they will still be used. We will mostly change the renderer down at `ispc` level.

>>> MyRenderer.ih

This file contains the declaration of the `ispc` structure and data used.

Our renderer used most of the same methods as the super class and it keeps no data, therefore:

```
#include “ospray/SDK/render/Renderer.ih

struct MyRenderer
{
  /*! the base "Renderer" we are derived from */
  Renderer super;
};
```

>>> MyRenderer.ispc

This file contains the definition of the `ispc` methods and data used.

The function, `MyRenderer_create()`, used to create and register the new `ispc` renderer (see the constructor of `MyRenderer` class in `MyRenderer.cpp`) is defined here (its name could be any, but the same as used in the `.cpp` file).

```
export void *uniform MyRenderer_create (void *uniform cppE)
{
    uniform MyRenderer *uniform self = uniform new uniform MyRenderer;
    Renderer_Constructor(&self->super,cppE);

    int const select_renderer = 0;  // set this to select one of the available renderers

    switch (select_renderer) {
       case 0:          // patterned pixels
           self->super.renderSample = MyRenderer_renderSample_testFrame;
           break;
       case 1:          // depth renderer
           self->super.renderSample = MyRenderer_renderSample_depth;
           break;
       default:
           self->super.renderSample = MyRenderer_renderSample_testFrame;
           break;
       }

    return self;
}
```

Note that besides calling the `ispc Renderer_Constructor()` this function also sets the pointer to the `renderSample()` method (if it didn't the default one, supplied with the `Renderer` class and corresponding `ispc` code would be used). This method will be responsible to process (shade) each sample in the image plane. In most cases one sample in the image plane corresponds to one pixel.

A `switch () { case: … }` statement is included conditioned on the local variable `select_renderer`, to allow us to use different `renderSample` functions, without much code changes.

Finally, our new `renderSample` functions are defined (there are two: `MyRenderer_renderSample_testFrame` and `MyRenderer_renderSample_depth`).

`MyRenderer_renderSample_testFrame` just sets the pixel color (`sample.rgb`) based on the sample ID. If you have a look at the `ScreenSample` struct definition in `Renderer.ih` (this is OSPRay core, remember) you will find:

```
struct ScreenSample {
  // input values to 'renderSample'
  vec3i sampleID; /*!< x/y=pixelID,z=accumID/sampleID */
  Ray   ray;      /*!< the primary ray generated by the camera */
  // return values from 'renderSample'
  vec3f rgb;
  float alpha;
```

```
  float z;
  vec3f normal;
  vec3f albedo;
};
```

Our code is as follows:

```
void MyRenderer_renderSample_testFrame(Renderer *uniform _self,
                                       FrameBuffer *uniform fb,
                                       World *uniform model,
                                       void *uniform perFrameData,
                                       varying ScreenSample &sample)
{
  sample.rgb.x = ((sample.sampleID.x)%256)/255.f;
  sample.rgb.y = ((sample.sampleID.y)%256)/255.f;
  sample.rgb.z = ((sample.sampleID.x+sample.sampleID.y+sample.sampleID.z)%256)/255.f;
  sample.alpha = 1.f;
  sample.z = 1.f;
}
```

OK! So this is it. Making sure that `select_renderer = 0` in `MyRenderer_create()` rebuild this module:

```
cd ospray-vi2/build
cmake .. -DCMAKE_INSTALL_PREFIX=../local-thrparty/install
make
```

Execute the example viewer:

```
> ./cornell_VI2
```

Using the mouse change the renderer on the menu and select '`myrenderer`'. You can afterwards hide the menu by pressing '`g`'.

The "depth" renderer

The previous renderer doesn't even shot a single ray – not much for a high performance ray tracing engine, such as `Embree` which lies at the foundations of OSPRay. Let us add a new `renderSample` method, which shoots a primary ray (ray from the camera into the scene, through the image plane) and returns its length – do you understand that the length of the primary ray is equal to the depth of the scene at the image plane point pierced by the primary ray?

If you look at the definition of a `Ray` in `ospray\common\Ray.ih` you find:

```
struct Ray {
  /* ray input data */
  vec3f org;  /*!< ray origin */
  float t0;   /*!< start of valid ray interval */
```

```
  vec3f dir;  /*!< ray direction */
  float time; //!< Time of this ray for motion blur

  float t;    /*!< end of valid ray interval, or distance to hit point after 'intersect' */
  int32 mask; //!< Used to mask out objects during traversal
  int32 rayID;
  int32 flags;

  /* hit data */
  vec3f Ng;    /*! geometry normal. may or may not be set by geometry intersectors */

  float u;     //!< Barycentric u coordinate of hit
  float v;     //!< Barycentric v coordinate of hit

  int primID;  //!< primitive ID
  int geomID;  //!< geometry ID
  int instID;  //!< instance ID
};
```

So `t` is the ray's length. We can use OSPRay `traceRay(model, ray)` method to shoot the primary ray and get the required returned data in the ray payload (data fields within the corresponding `Ray` struct). In `MyRenderer.ispc` you will find:

```
/*! renders the depth of the primary ray */
void MyRenderer_renderSample_depth(uniform Renderer *uniform _self,
                                   FrameBuffer *uniform fb,
                                   World *uniform model,
                                   void *uniform perFrameData,
                                   varying ScreenSample &sample)
{
  uniform MyRenderer *uniform self = (uniform MyRenderer *uniform)_self;

  traceRay(model,sample.ray);
  sample.z     = sample.ray.t;
  sample.alpha = 1.f;
  float normalizing_const = 1.e4;
  sample.rgb = make_vec3f(sample.z/ normalizing_const);
}
```

Make sure that `select_renderer = 1` in `MyRenderer_create()` and rebuild this module:

```
cd ospray-vi2/build
cmake .. -DCMAKE_INSTALL_PREFIX=../local-thrparty/install
make
```

Execute the example viewer and using the mouse change the renderer on the menu and select '`myrenderer`'. This is the same renderer you used in Tutorial 0.

That's all, folks!

Universidade do Minho