Hell Trace manual
By Przemysław Pastuszka

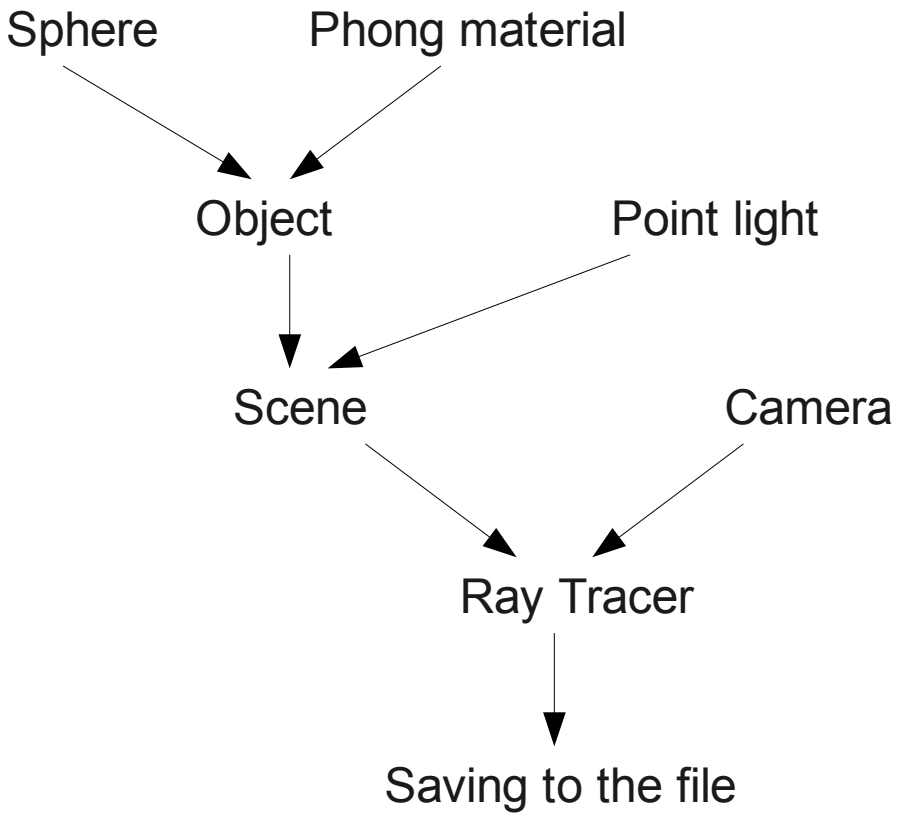
This manual has been written for Linux operating system. Obviously, HellTrace is working properly on Windows (other systems haven't been tested), but it is likely, that some of actions described below should be performed in other way there.

At the beginning you should compile source files using *makefile*. It is done by simply invoking *make* command. As the result the *helltrace* file should appear. You can execute it, giving input file name (with list of the tasks) as an argument: *./helltrace tasks_list*

But where to get file with the list of tasks? You can download sample one from the site of the project, but it will be better and more interesting to create it by yourself. Now it's the time to describe, how to do it.

The first important thing is to answer the question: what are those, mentioned tasks? You can perceive them as orders to execute some subprograms of HellTrace. This kind of orders also specify how this subprogram should work (by giving some attributes) and what other tasks should be finished previously. When the subprogram finishes, new resource is created in the environment (with one exception which is 'save to file' task), for example new pixel buffer or new light, and it can be used by other tasks.

List of tasks can be also presented as a dependency graph. For example:



It can be easily seen that first task to be done is 'Sphere'. It will execute appropriate subprogram, which creates sphere (with specified attributes) in the enviroment. Same thing with 'Phong material'. More interesting is 'Object',

because it produces object which agregates existing sphere and material. That is why 'Object' must be executed after 'Sphere' and 'Phong Material'.

There is none specific order in which tasks in the list should appear, because tasks manager will sort them topologically anyway. It is also remarkable, that only tasks, which are used to generate output image, will be executed. In other words: in dependency graph there must be a path from the task to 'Save to file' task, if you want it to be executed.

In input file you can use comments in C++ style like

//comment

or

/ comment*
comment/*

This is how description of the task should look:

TaskType TaskName
{
[input] AttributeType1 AttributeType1;
...
[input] AttributeTypeN AttributeTypeN;
}

TaskType specifies name of subprogram to be executed and *TaskName* is the name of resource, which will be generated by this task. In the brackets there is a list of attributes with optional *input* keyword. If there is an i*nput* before attribute *x* then resource *attributeNameX* is required to perform *TaskName* task. Some of the attributes can be omitted - then the default values will be used.

Here's the list of avalible task types, including attributes and description.

-**Sphere** - creates primitive, sphere. Attention: primitive is not yet an object, which could be used in a scene!
Attributes:
*origin - origin of sphere. Sample: *origin 0,2,1;*
*radius
-**Plane** - plane primitive
*normal - normal vector
*distance - distance from (0,0,0) point
-**Phong** - new material using blinn-phong shading
*color - material color (white is 1,1,1)
*specColor - specular color (default: 1,1,1)
*diffuse - diffuse coefficient
*reflectance - reflectance coefficient
*specPower - defines shape of specular (default: 100)
-**PointLight**
*color
*position
*primitive - point light can be also rendered like an usual object, but if so,

primitive must be specified (we are using existing resource - don't forget the *input* keyword!). If this attribute is omitted, point light will only light the scene
-**AreaLight**
*color
*primitive
*points - area light is approximated with a set of point lights. More point lights, better scene lighting, but also worse performance. Using this attribute you can specify exact number of points area light will be approximated with
-**Simple** - just an object to be rendered
*primitive
*material
-**Scene**
*size
*object1
...
*objectN - attributes from object1 to objectN specifies the objects the scene will contain
-**ConicCam**
*width - in pixels
*height - in pixels
-**RayTracing -** engine, produces pixel buffer
*camera
*scene
*tracingDepth - how many times the ray can be reflected (default: 5)
*width
*height
-**ImageLoader -** creates pixel buffer using a file
*fileName
*format - supported formats: raw (in this case you need also to specify width and height), bmp and hlt (own file format). Default: hlt
*width
*height - needed only when raw format is chosen
-**ImageSaver** - saves pixel buffer to a file
*buffer - input buffer
*fileName
*format - check ImageLoader
-**SimpleToneMap** - uses photo exposure to create new buffer
*buffer - input buffer
*exposure - (default: 1)
-**AdvToneMap** - uses Reinhard operator
*buffer
*middleGrey - (default: 0.6)
*whitePoint - (default: 16)
-**GaussianBlur** - uses Gauss filter
*buffer
*sigma - standard deviation

-**Lerp -** linear interpolation, blends two buffers
*buffer1
*buffer2
*value - should be from [0,1] range. Specifies how to blend buffers
-**DepthAntialiasing -** uses depth information to perform antialiasing
*buffer
*treshold - how big should be ratio between two depths to perform antialiasing(default: 1.2)
-**BrightPass -** compresses dark pixels
*buffer
*middleGrey
*whitePoint - check AdvToneMap
*offset - default 1
*treshold - default 0.5