LRay: A Lua Scriptable Ray Tracer

Simon Broadhead

December 15, 2011

This document presents the LRay ray tracing system, which was written as a term project for the Computer Graphics CS3GC3 course at McMaster University for the Fall 2011 semester.

Introduction

Author's Note

I wrote LRay over the course of five weeks in November and December 2011. While it is a course project, it was also a hobby project, and a platform for experimentation. It is more elaborate that I anticipated (or was asked for) but it I think it was worth the time. At various points in the project, I have experimented with embedded languages (namely Lua), Flex and Bison for scene definition (in a former revision, before Lua entered the picture), parallelism, and, of course, various ray tracing techniques. I intend to continue working on it in the future as time permits.

Technical Information

LRay was written in C++ using Microsoft Visual Studio 2010. It relies on the Boost[1] library for cross-platform threading and file-system capabilities, as well the Lua[2] and LuaBind [3] libraries for embedding Lua in a C++ application.

[1] `http://www.boost.org/`

[2] `http://www.lua.org/`

[3] `http://luabind.sf.net/`

LRay has only been tested on Windows, but should, with some simple modifications, be able to run on any platform. It has been tested and confirmed working on a fresh installation of Windows 7 after installing only the Microsoft Visual C++ Redistributable package.

About LRay

As opposed to other ray tracers, in which scenes are defined through a text-based description language, LRay uses Lua scripts for its scene definitions. There are numerous benefits to this approach, the most obvious being that procedurally generating geometry becomes trivial. Furthermore, animation becomes as simple as introducing a clock variable into the scene. Listing 1 shows an example of both procedural geometry and animation, and Figure 1 shows one frame of the output.

LRay is still very simplistic and not rigorously tested. It supports a small number of primitives, a single lighting model, a few procedural textures, and only a single type of camera. However, there is room for extensibility, and in the future, more features will be added.

```
scene_info = SceneInfo() { frames = 20 }
function scene(clock)
    local camera = Perspective(45) { transform =
        Rotate(y, 360*clock) * Translate(-5*z) }
    local s = Scene() { camera = camera }
    for i = 1,20 do
        local tf = Rotate(z, (i/20)*360) * Translate(x)
            * Scale(.1, .1, .1)
        local sphere = Sphere(tf)
        s:add_entity(sphere)
    end
    s:add_light(Light() { transform = Translate(x - 3*z) })
    return s
end
```

Listing 1: An example of procedurally generated geometry with animation. Note the use of the `clock` argument in controlling the position of the camera. This scene generates a ring of spheres and rotates the camera around it.

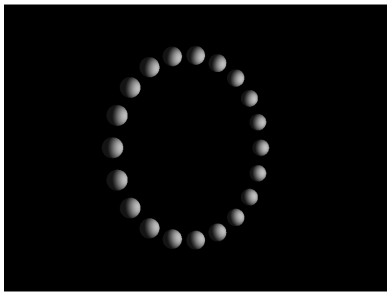
Figure 1: One frame of the scene generated by Listing 1

Usage

LRay is a command line application, and so must be invoked with command line arguments in order to run. It is invoked in the following way:

```
raytrace.exe <infile> <outdir>
```

The `infile` argument is the path to the Lua script that will be used to generate the scene. The `outdir` argument specifies the directory that will receive the rendered Targa images.

`raytrace.exe` must be invoked from the directory that contains its support files (*e.g.*, `luascene.lua`), otherwise it will not be able to continue.

Scene Files

Scene files are written entirely in the Lua programming language. An online reference for the language can be found at `http://www.lua.org/pil/index.html`. Only a basic knowledge of the language is necessary to create scene files for LRay.

Although Lua is not an object-oriented programming language, the conventions adopted by the LRay support library allow us to behave as though it were, referring to objects and classes as though they were part of the language. In this sense, LRay uses a *domain-specific language* embedded in Lua, which may differ in usage from every-day Lua.

Classes in the LRay object model are created by invoking their constructors, which are functions with the same name as the class and may have zero or more arguments relating to the instantiation of the object. Their attributes can either be set with the usual dot style familiar from other languages, or they can be set in "bulk" by placing the name of the instance to the left of a set of attributes and their values, wrapped in braces. Listing 2 demonstrates these possibilities.

```
local l = Light() { color = white, area = true }

-- Is the same as...

local l = Light()
l.color = white
l.area = true
```

Listing 2: Two examples of setting the attributes of an LRay object.

There are two things that should be defined by the Lua script in order to constitute a valid LRay scene file:

- A function called `scene` that accepts a single argument called `clock`, and returns an object of type `Scene`.
- A global variable called `scene_info` of type `SceneInfo`.

Scene Info

The type `SceneInfo` contains attributes that control how the scene will be rendered. If the global variable `scene_info` is not set then some default settings will be used.

`SceneInfo` ATTRIBUTES

`image_width`	*int*	The width in pixels of the output image.
`image_height`	*int*	The height in pixels of the output image.
`frames`	*int*	The number of frames of animation to render.
`samples`	*int*	The level of super sampling to use for anti-aliasing. For every additional level of super sampling, rendering time will approximately quadruple.
`start_clock`	*real*	The value of the `clock` argument to the `scene` function in the first frame.
`end_clock`	*real*	The value of the `clock` argument to the `scene` function in the last frame. The value of `clock` is linearly interpolated between `start_clock` and `end_clock` over the course of the animation.

Scene

The type `Scene` holds all the information about what will be rendered, including geometric entities, lights, and camera. Its constructor takes no arguments.

`Scene` ATTRIBUTES

`background_color`	*`Color`*	The color that is rendered when a ray does not intersect with any geometry in the scene.
`camera`	*`Camera`*	The camera from which rays are cast into the scene.

`Scene` METHODS

`add_entity(entity)`	Add a geometric entity (of type `Entity` or `Mesh`) to the scene.
`add_entities(entities)`	Add an array of geometric entities to the scene.
`add_light(light)`	Add a light (of type `Light`) to the scene.
`add_lights(lights)`	Add an array of lights to the scene.

Camera

So far, there is only one type of camera available, the `Perspective` camera. Its constructor accepts up to two arguments. The first argument is the horizontal field of view in degrees, and the second argument is the aspect ratio. If omitted, the default values are 45 and 1.33 respectively. The untransformed camera is placed at the origin looking down the positive z-axis.

`Perspective` ATTRIBUTES

`transform`	*`Transform`*	The affine transformation that moves the camera from its default position.

Entity

An entity is the main source of geometric data in a scene (the other being a `Mesh`, which is a collection of triangular entities). The `Entity` is a base class, and has no constructor of its own, but all entities derive from `Entity` and thus share its attribute. Currently, `Entity` has only one attribute.

The following are the types of entities that are available in LRay. They all accept a single argument in their constructor, a `Transform` which moves the entity from its default position to the desired position in the world. Figure 2 shows the default sizes of the primitive shapes.

- `Sphere`. A unit sphere centered at the origin.
- `Cylinder`. A cylinder with radius 1 and height 1, extending upward from the origin.
- `Cone`. A cone with radius 1 and height 1, extending upward from the origin.
- `Cube`. A unit cube centered at the origin.

`Entity` ATTRIBUTES		
`texture`	*`Texture`*	The texture that defines how the geometry appears when it is rendered.

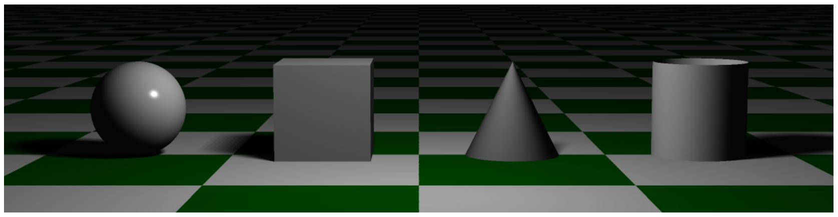
Figure 2: The four primitive shapes in LRay, along with a `Floor` mesh.

Planes are no longer supported by LRay due to their infinite bounding box which severely hinders the acceleration techniques used internally.[4] There is a convenience function called `Floor` which takes no arguments and generates a large flat mesh that resembles a plane from a low enough camera position, suitable for simple renderings.

[4] LRay uses *kd-trees* internally, which rely on wrapping a bounding box around the entire scene and splitting it up in an efficient way. If the bounding box is infinite, the algorithm fails to provide any speed-up.

Mesh

The `Mesh` class defines a collection of triangular entities with shared vertices. They consist of a list of vertices, optional lists of normal vectors and *uv* (texture) coordinates, and a list of indices that define which vertices, normals, and *uv* coordinates should be used for each face. If normal vectors are not supplied, then they are computed automatically by LRay, and if *uv* coordinates are not supplied, then every individual triangle has the entire texture mapped to it.

Unlike other places where points and vectors are declared, `Mesh` does not use the `Vector` type. For brevity, all vectors in a `Mesh` definition are simply arrays of three real numbers.

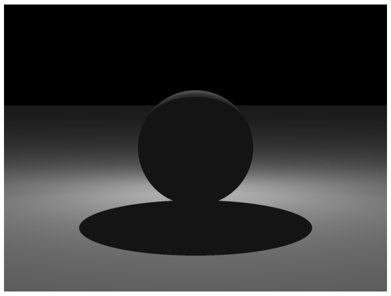
Figure 3: A scene rendered with point light. Rendering took 0.62 seconds.

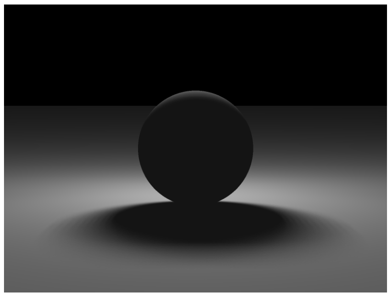
Figure 4: A scene rendered with a 10 by 10 area light. Rendering took 18.59 seconds.

Light

Every scene should have at least one `Light`, otherwise everything will appear very dark and flat. A light in LRay can be either a simple point light, or an area light. A point light is very fast to render, but produces completely sharp shadows. An area light produces soft shadows, but dramatically increases rendering time.

An area light is a rectangular grid of light cells, (by default oriented along the *xz* plane) and each cell contains a light source. The more cells there are, the higher the quality of shadows, but the longer it will take to render.

`Mesh` ATTRIBUTES

`vertices`	*{{x,y,z}}*	An array of points (each represented as an array of three real numbers), each giving the position of a point on the mesh
`normals`	*{{x,y,z}}*	An array of vectors (each represented as an array of three real numbers), each defining a normal vector for the mesh
`uvs`	*{{u,v}}*	An array of coordinates (each represented as an array of two real numbers), each defining a *uv* texture coordinate for the mesh
`indices`	*{{v,u,n}}*	An array of face indices, each one of which is an array containing one, two, or three elements, depending on whether normals or *uv* coordinates were specified. The second element must always be *uv* and the third must always be normals, so if only vertices and normals were specified, then the second index should be `nil`. If both normals and *uv*s are omitted, then the array may be replaced by a single integer.
`transform`	*Transform*	The affine transformation that moves the mesh from its default position.
`smooth`	*bool*	If true, the normal vectors will be interpolated between points, giving the appearance of a smooth object when struck by light. Otherwise, the mesh will have a faceted appearance.

`Mesh` METHODS

`copy()`	Return a copy of a mesh, so it doesn't need to be loaded from a file again.

```
function Floor()
    return Mesh() {
        vertices = {
            {-100.0, 0.0, -100.0},
            {100.0, 0.0, -100.0},
            {100.0, 0.0, 100.0},
            {-100.0, 0.0, 100.0},
        },
        uvs = {
            { -25.0, -25.0 },
            { 25.0, -25.0 },
            { 25.0, 25.0 },
            { -25.0, 25.0 }
        },
        indices = {
            { {1, 1}, {4, 4}, {2, 2} },
            { {2, 2}, {4, 4}, {3, 3} }
        }
    }
end
```

Listing 3: The definition of the `Floor` function. Note that *uv* coordinates outside the $[0,1]$ range correspond to repetition of the base pattern.

.

`Light` ATTRIBUTES

`color`	*`Color`*	The colour that this light will cast onto the scene
`rows`	*int*	The number of rows in the area light grid.
`cols`	*int*	The number of columns in the area light grid.
`area`	*bool*	If true, every time a light cell is sampled, the effective position of the light source will be jittered randomly within the cell. This prevents soft shadows from having an obvious "banding" effect.
`transform`	*`Transform`*	The affine transformation that moves the light from its default position.

Texture

A `Texture` is an object that defines what colour will be visible at each point on an entity. A texture has two attributes, `finish` and `pigment`.

A finish is a collection of attributes that control how light reacts with the surface of an entity. It has the type `Finish`. A pigment is an object that maps each point on the surface of an entity to a colour. A pigment can be either 2-dimensional or 3-dimensional.

A 2-d pigment maps 2-d points on the surface to colours (*e.g.*, as in a checkerboard pattern pasted on the surface of a sphere), while a 3-d pigment maps 3-d points in space to colours (*e.g.*, as in a sphere carved out of a checkerboard block).

Each pigment has a texture map associated with it as well which specifies how the coordinates on an entity should be mapped to coordinates on the texture.

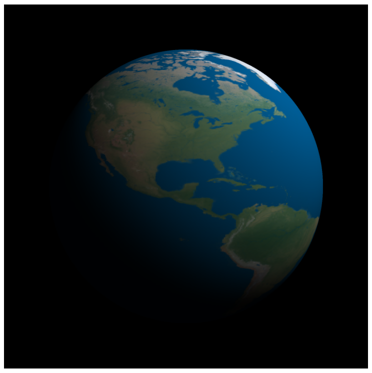
Figure 5: An example of the `ImagePigment2D` pigment.

`Texture` ATTRIBUTES

`finish`	*`Finish`*	The finish that affects how light interacts with this texture.
`pigment`	*`Pigment`*	The pigment that affects the colour of each point on the texture.

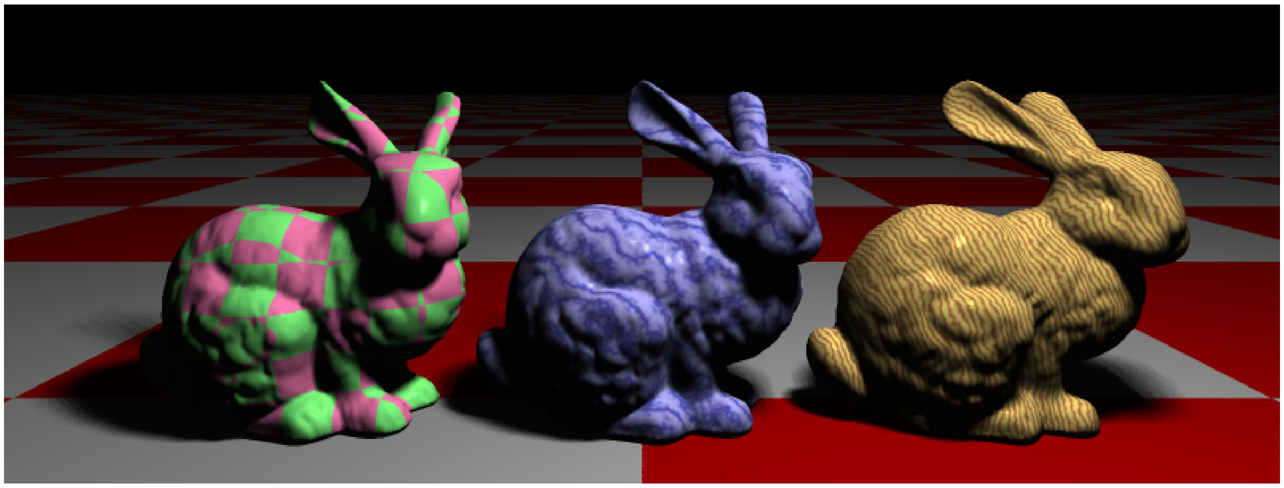
Figure 6: The `CheckerPigment3D`, `MarblePigment3D`, and `WoodPigment3D` pigments in action, with a `CheckerPigment2D` on the floor.

Finish ATTRIBUTES

ambient	*Color*	Specifies the colour that should be drawn when no light hits the texture. Realistically, it should be zero, but very dark greys can give more definition.
diffuse	*real*	The percentage of colour that should be contributed by diffuse light scattering.
specular	*real*	The percentage of colour that should be contributed by specular highlight.
exponent	*real*	The Phong exponent for the specular highlight. Larger values give a smaller, tighter highlight.
reflection	*real*	The percentage of colour that should be contributed by reflection of the surrounding environment.
refraction	*real*	The percentage of colour that should be contributed by refracting through the surface and out the other side.
refraction_index	*real*	The index of refraction to use for the surface material.
absorption	*real*	The percentage of colour that should be absorbed due to refraction (giving a "cloudy" appearance).

Pigment

There are six pigments currently supported by LRay: `ColorPigment`, `CheckerPigment2D`, `CheckerPigment3D`, `WoodPigment3D`, `MarblePigment3D`, amd `ImagePigment2D`.

The simplest is `ColorPigment` which is simply a constant colour, passed into its constructor.

`CheckerPigment2D` and `CheckerPigment3D` function identically aside from the way they interpret texture coordinates. They both alternate between two other pigments in a checkerboard fashion.

CheckerPigment2D AND CheckerPigment3D ATTRIBUTES

pigment1	*Pigment*	The primary pigment in the checker pattern.
pigment2	*Pigment*	The alternate pigment in the checker pattern.
texture_map	*TextureMapxD*	The texture map which converts object coordinates into texture coordinates.

`WoodPigment3D` and `MarblePigment3D` also function very similarly, in that they are both Perlin noise-based textures with the same attributes. The difference is that while `MarblePigment3D` simulates a marble pattern with stripes, `WoodPigment3D` simulates wood with rings about the z axis.

Finally, the `ImagePigment2D` allows the mapping of 2-d raster images onto entities. It accepts a single mandatory argument to its constructor, the filename of the texture to load (which should be in the same directory as the Lua script). Presently the only supported file

`WoodPigment3D` AND `MarblePigment3D` ATTRIBUTES

`color1`	*Color*	The primary colour in the texture.
`color2`	*Color*	The secondary colour in the texture.
`frequency`	*Vector*	A vector containing the number of repetitions of the base pattern there should be in each direction.
`power`	*real*	The intensity of the Perlin noise perturbation of the base pattern.
`size`	*real*	The size of the minimum Perlin noise harmonic.
`texture_map`	*TextureMap3D*	The texture map which converts object coordinates into texture coordinates.

format is uncompressed Targa, either 24- or 32-bit. The only attribute of `ImagePigment2D` is `texture_map`, a `TextureMap2D`.

Texture Map

All 2-d and 3-d pigments have an attribute called `texture_map` which is of type either `TextureMap2D` or `TextureMap3D`. The former takes uv coordinates on the surface of an entity and translated them into 2-d texture coordinates. The latter takes standard coordinates in $\mathbb{R}^3$ and maps them to 3-d texture coordinates.

The default instances of these two classes simply map the input coordinate directly to the output coordinate without any changes. Presently the only other texture maps available are simple linear transformations of the input coordinates. The 2-d version is called `UVMap` and simply allows for scaling and translating the input coordinates. The 3-d version is called `LinearMap` and allows for applying a 3-d transformation to the input coordinates, exactly the same way as applying a transformation to an entity. Note that in both cases it is the coordinates that are transformed and not the texture. Thus, to make a texture smaller, the scale factor should be positive, and to move a texture to the left, the coordinates should be moved to the right.

`UVMap` ATTRIBUTES

`scale_s`	*real*	The amount of scaling in the s (horizontal) direction.
`scale_t`	*real*	The amount of scaling in the t (vertical) direction.
`translate_s`	*real*	The amount of translation in the s (horizontal) direction.
`translate_t`	*real*	The amount of translation in the t (vertical) direction.

`LinearMap` ATTRIBUTES

`transform`	*Transform*	The transformation to apply to the input coordinates.

More sophisticated mappings are possible, such as spherical or cylindrical mapping, but they are not presently implemented in LRay.

Vectors and Colours

The `Vector` class represents a single coordinate in $\mathbb{R}^3$. It has the usual mathematical operators defined on it, so basic vector arithmetic is possible. They can be added, subtracted, negated, and scaled as usual vectors. Vectors are given with the `Vector` constructor, as in `local pos = Vector(1.0, 2.0, 3.0)`. For convenience, the standard unit vectors are defined with the names `x`, `y`, `z`, and the zero vector has the name `origin`.

The `Color` class represents a single RGB colour. Presently there are no mathematical operators defined on colours. A few standard colours are defined as well: `black`, `white`, `red`, `green`, and `blue`.

`Vector` ATTRIBUTES

`x`	*real*	The x component of the vector.
`y`	*real*	The y component of the vector.
`z`	*real*	The z component of the vector.

`Color` ATTRIBUTES

`r`	*real*	The red component of the colour.
`g`	*real*	The green component of the colour.
`b`	*real*	The blue component of the colour.

Transformations

The `Transform` class is used to do linear and affine transformations on points in the scene. Every `Entity` accepts a `Transform` in its constructor, and `Light`, `Camera`, and `TextureMap3D` have transformations as well.

A `Transform` can do two things. First, it can be multiplied by another `Transform` to give a composite transformation. Second, it can return its inverse, by calling its `inverse` method.

`Transform` METHODS

`inverse()`	Return the inverse of a transformation.

There are four different functions which construct `Transform`s.

- `Translate(vector)`. Accepts a translation vector as its sole argument, and returns a transformation that translates in the given direction.
- `Scale(x, y, z)`. Accepts three scaling factors—one for each axis—and returns a transformation that scales by the specified amounts.

- `Rotate(vector, angle)`. Accepts an axis and angle of rotation and returns a transformation that rotates by the specified number of degrees around the given axis.
- `LookAt(pos, at, up)`. A combination of translation and rotation, this function accepts a position vector, a "look-at" vector, and an up orientation, and returns an orthogonal transformation such that the origin moves to the `pos` vector, and the point `at` lies on the z axis. The `up` vector must not be parallel to the `at` vector.

Note that LRay uses a *left-handed* coordinate system, so the positive z-axis is oriented into the screen.

Future Work

LRay in its present state is a very simple ray tracer. More sophisticated material properties such as refractive internal media, glossy finishes, and subsurface scattering are possible areas of improvement. Global illumination would be an excellent addition, though also very labour consuming to implement.

A preliminary implementation of constructive solid geometry (CSG) for creating complex shapes out of simple primitives was completed, but not working well enough for project submission, so it was removed. Restoring and improving it is another area of future improvement.

Compilation Notes

In order to compile this project, the Boost library *must* be installed. It can't be provided with the source tree due to the way it is included in projects.

It can be downloaded for Windows from `http://www.boostpro.com/download/`. LRay was developed with Boost 1.47.0. The basic header files as well as Boost Thread, Boost System, Boost DateTime, and Boost Filesystem are the only libraries that need to be installed (choose all variants for your Visual Studio version to be safe). The rest (particularly Boost Wave, since it is enormous) can be omitted.

Once Boost is installed, the library and include directories will need to be added to Visual Studio's include and library directories.

Lua and LuaBind are included in the project source tree, so they should not require any additional setup.

The project was developed on Visual Studio 2010, but a Visual Studio 2008 project is included. The executable has been verified fully working on a fresh Windows 7 installation with nothing installed but the Microsoft Visual C++ Redistributable.