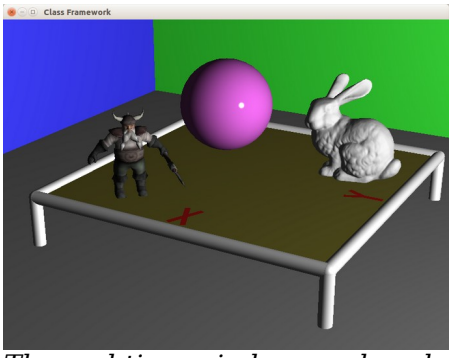
CS500 Project 0
(ungraded)

Synopsis

Prepare your framework for a semester of raytracing. The framework, as distributed, reads and parses a scene definition, and then provides a real-time interactive display of the contents. Your goal is to remove all real-time graphics, and replace it with code that stores the parsed information in preparation for the actual raytracing which will commence with project 1.

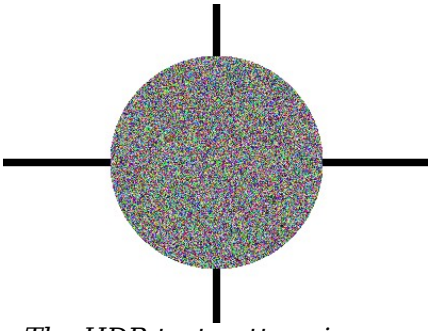


The real-time window produced by the original framework. You may remove this code.

Description

The framework contains a number of features which you can use to get started on the implementation of the raytracing algorithm quickly and as distraction-free as possible. This includes:

- A parser for our scene-description language (**ReadScene** in **main.cpp**). This is easy to extend as the semester progresses.
- An HDR output file format (called *RGBE* or *RADIANCE*), and code (in **rgbe.cpp**) to read and write such images . This will serve as the main output of the raytracer. There are several ways to view such images:
  - An open-source app named **luminance-hdr** (see http://qtpfsgui.sourceforge.net/) can display this and other formats of HDR images and perform extensive tone-mapping on them.
  - Recent versions of Windows seem to have the ability to display such images with a simple double-click on the filename.
  - I may also supply a small convenient Python app to do the same. It has several features especially suited to this class.
- Code to generate and write a simple test image (seen at right). This will be replaced by your actual raytracing code.
- A single line **#pragma** for **OpenMP** (http://openmp.org/wp/) that *magically* turns any C++-loop into a multi-threaded loop.
- An open-source library named **GLM** for all vector, matrix, and quaternion calculations.
- An open-source template library called **bvh** to provide a spatial data structure for project 1.



The HDR test pattern image produced by the framework. You will replace this with a raytraced image.

- **Assimp**, the **Open Asset Import Library** (http://assimp.sourceforge.net/), to read in mesh type models from any of 30+ file formats, and code (in **readAssimpFile.cpp**) that extracts the geometry and some material information from the resulting structure.
- A choice of a good quality random number generator (a *Mersenne Twister* implementation called *MT19937-64*) which is thread-safe (critical when using OpenMP). Call **myrandom(RNGen)** to get a random number uniformly distributed in the range $[0.0, 1.0]$.

Instructions

The framework, as provided, is a real-time application that reads the scene and displays it with mouse/keyboard viewing/navigation interactions. To get the bare-bones starting point for your raytracer, remove all the real-time drawing code. The remaining husk of an application has all the features mentioned in the above bullet list. Alternately, in recent semesters, several students have found it convenient to keep both the real-time display and the (distinctly non-real-time) raytracer code in the same application and provide some keystroke to start the raytracer running with the transformations currently producing the real-time image.

Identifying all the real-time code has been made purposely easy:

- Files **realtime.h** and **realtime.cpp**, define a class **Realtime** containing all the real-time code.
- Shader files **realtime.vert** and **realtime.frag** are used by the real-time code.
- File **raytrace.h**, includes **"realtime.h"** and declares **Realtime* realtime**.
- File **raytrace.cpp**, constructs the **Realtime** instance in **Scene::Scene()** .
- All other references to the “real-time” structure (all found in **raytrace.cpp**) are method calls on the **Realtime** structure for passing information about the scene as it's parsed. They mostly convert that information into OpenGL calls and pass it on to the GPU.

Details

The input file format consists of individual lines, each of which consists of a keyword followed by a number of parameters, usually all floats, but occasionally containing some strings. Each line parsed from the scene file will be passed to the **Scene** structure through a call to **void Scene::Command(const std::vector<std::string>& strings, const std::vector<float>& f)** Its arguments are (1) the input line parsed into a std::vector of space-separated words, and (2) a std::vector of floats containing each of those strings parsed as a float (or a zero if such parsing is not possible).

Your implementation of **Scene::Command** should examine the first word (in strings[0]) to determine what the line's command is, grab the command's parameters from the vector of floats or (rarely the vector of strings), and store that information somewhere in the **Scene** structure. The **Scene** structure should be able to store the following data:

- Screen **width** and **height**
- Camera **eye**, **orientation** and **ry** information.
- A list of materials (either normal reflecting surface types or light emitting surface types).
- A list of geometric shapes (box, sphere, cylinder, and triangle to start with, but probably extend later). Each shape will be assigned the most recently defined material.
- A list of lights. That is, objects whose assigned material is a light emitter.

The (starting) list of commands and their parameters are:

- **screen** width height
- **camera** x y z ry <orientation> // Eye vector, viewing ratio, orientation spec
- **ambient** r g b // Ambient light color
- **brdf** r g b r g b alpha // Diffuse color, specular color, shininess
- **light** r g b // Light color/brightness
- **sphere** x y z r // Center and radius
- **box** bx by bz dx dy dz // Base corner point and diagonal vector
- **cylinder** bx by bz ax ay az r // Base point, axis vector, radius
- **mesh** file tx ty tz scale <orientation> // Mesh filename to read, translation, scale, orientation
- The two <orientation> parameters above indicate a list of concatenated rotations specified in a mini-language consisting of any sequence of rotations specified in these forms:
  - **x** angle, **y** angle**,** or **z** angle**:** for rotations around the canonical bases,
  - **a** angle x y z: for angle-axis rotations, and
  - **q** qw qx qy qz**:** for a quaternion.
- A procedure, **Orientation,** is supplied to parse these orientation specifications into a single quaternion.

The following structures and functions should be considered:

- **Camera:** A camera object will contain the eye point, the view orientation, and the r_y ratio, and will be able to generate rays for all the pixels on the screen. (More generic camera models could be added later, but this is sufficient for the course.)

- **Ray**: A class defining a ray, initialized with a point and a unit length direction. It may be convenient to implement an **eval** method to compute a point on the ray from a t parameter value.

- A base class **Shape,** and a number of sub-classes **Sphere**, **Box**, **Cylinder**, **Triangle,** etc. Each sub-class will provide a (virtual) **intersect** method to compute the intersection of a ray with the shape. Every **Shape** also needs to be created with a pointer/reference to a **Material** instance.

- **Intersection**: A record of a ray's intersection with a shape. This structure should record which **Shape** was hit, the parameter value t of the intersection point, and the position and normal of the intersection. Later, if you intend to implement textures, this structure should also contain the texture coordinate of the intersection point.

- **Material**: A class for describing a **Shape**'s surface properties of reflecting or emitting light. Implement two sub classes inherited from this, one for surfaces that reflect light (named **BRDF** perhaps**,** with attributes for **diffuse** and **specular** color and shininess **alpha**) and another subclass for surfaces that emit light, (named **Light** perhaps, with an RGB attribute for emitted light). You will need a method to differentiate the two sub-types (perhaps **isLight** returning **true** or **false**).

- **Scene:** A structure to contain the screen **width** and **height**, one **Camera** object , a list of all **Shape** objects and another list of the just those shapes which emit light. (I suggest **vector<Shape*>** for both lists.)

- **TraceRay(Scene, Ray):** A function (or **Scene** method) which traces a single ray into the scene, tests for intersection with all objects in the scene and returns the front-most intersection (if any).

Test file testscene.scn

```
screen 400 300
camera 5.553228 2.942755 2.900874 0.2 q 0.416981 0.279589 0.480987 0.718247
ambient 0.125000 0.125000 0.125000
light 5 5 5
sphere 1.900000 5.000000 2.000000 2.0
brdf 0.600000 0.600000 0.600000 0.000000 0.000000 0.000000 10 0.000000 0.000000 0.000000 1.0
box -3.000000 -3.000000 -0.100000 18.000000 18.000000 0.100000
brdf 0.600000 0.600000 0.600000 0.000000 0.000000 0.000000 10 0.000000 0.000000 0.000000 1.0
box -3.000000 -3.000000 8.900000 18.000000 18.000000 0.100000
brdf 0.200000 0.900000 0.200000 0.000000 0.000000 0.000000 10 0.000000 0.000000 0.000000 1.0
box -3.000000 -3.000000 -0.100000 0.100000 18.000000 9.000000
brdf 0.900000 0.200000 0.900000 0.000000 0.000000 0.000000 10 0.000000 0.000000 0.000000 1.0
box 15.000000 -3.000000 -0.100000 0.100000 18.000000 9.000000
brdf 0.200000 0.200000 0.900000 0.000000 0.000000 0.000000 10 0.000000 0.000000 0.000000 1.0
box -3.000000 -3.000000 -0.100000 18.000000 0.100000 9.000000
brdf 0.900000 0.900000 0.200000 0.000000 0.000000 0.000000 10 0.000000 0.000000 0.000000 1.0
box -3.000000 15.000000 -0.100000 18.000000 0.100000 9.000000
brdf 0.595000 0.560000 0.210000 0.000000 0.000000 0.000000 200 0.000000 0.000000 0.000000 1.0
box -1.200000 -1.200000 0.390000 2.400000 2.400000 0.020000
brdf 1.000000 1.000000 1.000000 0.000000 0.000000 0.000000 20 0.000000 0.000000 0.000000 1.0
sphere 1.200000 1.200000 0.400000 0.05
cylinder 1.200000 1.200000 0.000000 0.000000 0.000000 0.400000 0.05
cylinder 1.200000 1.200000 0.400000 -2.400000 0.000000 0.000000 0.05
sphere 1.200000 -1.200000 0.400000 0.05
cylinder 1.200000 -1.200000 0.000000 0.000000 0.000000 0.400000 0.05
cylinder 1.200000 -1.200000 0.400000 -0.000000 2.400000 0.000000 0.05
sphere -1.200000 -1.200000 0.400000 0.05
cylinder -1.200000 -1.200000 0.000000 0.000000 0.000000 0.400000 0.05
cylinder -1.200000 -1.200000 0.400000 2.400000 0.000000 0.000000 0.05
sphere -1.200000 1.200000 0.400000 0.05
cylinder -1.200000 1.200000 0.000000 0.000000 0.000000 0.400000 0.05
cylinder -1.200000 1.200000 0.400000 0.000000 -2.400000 0.000000 0.05

brdf 0.900000 0.400000 0.900000 0.000000 0.000000 0.000000 2000
sphere -0.500000 -0.500000 0.800000 0.4

brdf 0.900000 0.100000 0.100000 0.000000 0.000000 0.000000 10 0.000000 0.000000 0.000000 1.0
mesh letterX.ply 0.975000 0.000000 0.420000 0.1  q 0.707107 0.000000 0.000000 0.707107
mesh letterY.ply 0.000000 0.975000 0.420000 0.1  q 1.000000 0.000000 0.000000 0.000000

brdf 0.900000 0.900000 0.900000 0.000000 0.000000 0.000000 10 0.000000 0.000000 0.000000 1.0
#mesh dwarf.x  0.800000 -0.600000 0.40000 0.01 q 0.271 0.271 0.653 0.653
#mesh bunny.ply  -0.500000 0.800000 0.2300 5.0  z -90 q 0 0 0.707107 0.707106
```