RenderPark source code structure

Philippe Bekaert, Frank.Suykens
Computer Graphics Research Group
Departement of Computer Science
K. U. Leuven, Leuven (Belgium)
{Philippe.Bekaert|Frank.Suykens}@cs.kuleuven.ac.be

November, 5 1999

Introduction

This document describes how the RenderPark source code is structured. It is meant as a starting point for people who wish to use RenderPark as a reference for their own implementations, or who consider to implement new (or old) rendering algorithms, using the large collection of "tools" that are offered in RenderPark. RenderPark contains a large number of recently and less recently proposed rendering algorithms. The source code structure reflects that these rendering algorithms are all built on top of a number of basic building blocks that are common for many algorithms. An example of such a basic building block is for instance the tracing of a ray in order to determine to nearest intersection point along the ray with the surfaces of the scene.

Why is collaborating in a project like RenderPark interesting?

- Sharing work is saving work: you don't have to do everything yourself as a lot things you need are already there, it is good for the quality of the implementations, allows to test each others work, it is an impulse to provide documentation (so others can use your work), leads to more careful design, a more efficient implementation of a common subtask is advantageous for all algorithms that are built on top of this subtask (e.g. ray-tracing acceleration schemes), ...;
- It makes a fair comparison of rendering algorithms possible since all algorithms are built upon a common "tool box" and use the same conventions ...
- It makes interesting demonstrations;
- RenderPark is a prototype rendering engine, that should be easy to tune for a particular application, and it can be a reference implementation for your rendering algorithms;
- It's good publicity for your work and will be better publicity as more people use it;
- ...

RenderPark is made by global illumination researchers and is also mainly targeted, but not restricted, to researchers in this exciting field.

This document contains two parts: first a part describing what can be found in which source file (§1), next a part describing the main data structures (§2). Section 3 and 4 present the actual implementation of rendering algorithms.

Chapter 1

Source code structure

1.1 Main directory

See figure 1.1. The main directory basically contains:

- the main program (`main.c`);
- routines for reading `MGF` and `VRML'97` scenes (`main, readmgf, readvrml, PhBRML`), building a scene data structure and auxiliary structures (see §2), saving an image in `PPM` or `TIFF` format, saving the "illuminated" scene model after radiosity computations in `VRML'97` format (`writevrml`)...
- everything of the graphical user interface that does not concern specific rendering algorithms (`ui_*`, X/Motif simplified with a small user interface toolkit in `uit`);
- drivers for graphics hardware (OpenGL, IrisGL and starbase), manipulation of the virtual camera using the mouse, object selection (to update material properties e.g.), ...(`render, render-common, canvas, camera, ...`). Rendering algorithms that use graphics hardware for visualisation of the result shall fill in the colour to be used for each patch or vertex in the patch and vertex data structures that are offered (`vertex_type.h, patch_type.h`);
- monitor calibration (to set monitor white point, gamma correction, and brightness adjustment) and tone mapping (`color.c, eye.c`);
- command line option processing (`options.*`), batch and offscreen rendering (`batch`), Inter Process Communication control (`ipc`);
- a "tool box" for rendering algorithms:
  - ray-object intersection routines (`geom, grid, patch, shaftculling, shadow-caching`);
  - numerical integration on the square, triangle, 3D cube (`cubature`);
  - ID-rendering, determination of directly received visual potential (`render, potential`),
  - uniform and $\cos\theta$ sampling of spherical triangles (`spherical`), or patches (`patch`)
  - KD-trees, transforms, line-plane and line-line intersection computation, ...;
  - ...
- hooks for rendering algorithms: we make a distinction between pixel-based rendering algorithms (such as stochastic ray tracing, `raytracing`) and world-space algorithms such as radiosity (`ra-diance`).

```
0README              bsdf_methods.h      lightlist.C         shadowcaching.h
ARCHIVE              btdf.c              lightlist.H         shaftculling.c
BREP                 btdf.h              lightmeter-
ing.c      shaftculling.h
Boolean.h            btdf_methods.h      lightmetering.h     shooting.rw
ChangeLog            bzero.c             main.c              spectrum.h
Config.Alpha         camera.c            material.c          spectrum_type.h
Config.FreeBSD       camera.h            material.h          spherical.c
Config.HP            canvas.c            materiallist.c      spherical.h
Config.Linux         canvas.h            materiallist.h      splitbsdf.c
Config.SGI           cie.c               monitor.c           splitbsdf.h
Config.Solaris       cie.h               monitor.h           starbase.c
Config.common        cluster.c           motif.h             statistics.h
Config.site          cluster.h           mymath.h            stratification.C
Config.site.default  color.c             namedvertex.c       stratification.H
DEST                 color.h             namedvertex.h       sub.sh
DOC                  color.h.antiek      opengl.c            surface.c
Float.h              compound.c          options.c           surface.h
GALERKIN             compound.h          options.h           topology.c
GDT                  cubature.c          packedcolor.c       topology.h
GENETIC              cubature.h          packedcolor.h       transform.c
HURA                 defaults.h          patch.c             transform.h
IMAGE                deps                patch.h             ui.h
MCRAD                edf.c               patch_flags.h       ui_camera.c
MCRAD.2              edf.h               patch_type.h        ui_canvas.c
MCRAD.stable         edf_methods.h       patchlist.c         ui_config.c
MGF                  error.c             patchlist.h         ui_debug.c
MGF-2.0a             error.h             patchlist_geom.h    ui_file.c
Makefile             eye.c               phong.c             ui_help.c
NON_PUBLIC           eye.h               phong.h             ui_main.c
PARTICLE             fileopts.h          polygon.h           ui_material.c
PHOTONMAP            fix.c               pools-1.4           ui_material.h
PNM                  fix.h               potential.c         ui_phong.c
POOLS                geom.c              potential.h         ui_phong.h
PhBRML.C             geom.h              radiance.c          ui_radiance.c
PhBRML.H             geom_methods.h      radiance.h          ui_raytracing.c
QMC                  geom_type.h         ray.h               ui_render.c
RAYTRACING           geometry3d.c        raycasting.h        uit.c
RenderPark           geometry3d.h        raytracing.c        uit.h
SCENES               geomlist.c          raytracing.h        vector.c
SE                   geomlist.h          readmgf.c           vector.h
SGL                  gl.c                readmgf.h           vectorlist.c
TODO                 grid.c              readvrml.C          vectorlist.h
TOOLS                grid.h              readvrml.H          vectoroctree.h
ad2c.script          gridP.h             render.h            vectortype.h
appdata.h            hemirenderer.C      rendercommon.c      vertex.c
batch.c              hemirenderer.H      renderhook.c        vertex.h
batch.h              hitlist.c           renderhook.h        vertex_type.h
bidir-lab.fig        hitlist.h           renderhook_priv.h   vertexlist.c
bidir-lab.fig.bak    hitlist_type.h      rgb.h               vertexlist.h
bounds.c             ipc.c               rtdummy.C           writevrml.c
bounds.h             ipc.h               rtdummy.h           writevrml.h
brdf.c               jacobian.c          scene.c             xxdf.c
brdf.h               jacobian.h          scene.h             xxdf.h
brdf_methods.h       kdtree.C            select.c
bsdf.c               kdtree.H            select.h
bsdf.h               lib                 shadowcaching.c
```


Figure 1.1: Files in the `RenderPark` main directory

1.2 subdirectories

- `DOC/`: documentation;
- `TOOLS/`: sub-directory for small test programs and tools for e.g. comparing images. Does not contain code that is needed to build the `RenderPark` executable
- Auxiliary libraries:
  - `GDT/`: generic data types (in C, to be superceeded by C++ sometimes in the future);
  - `POOLS/`: memory management;
  - `MGF/`: MGF parser library;
  - `SGL/`: Simple Graphics Library (for clustering in Galerkin radiosity methods e.g., also useful for other purposes);
  - `BREP/`: Boundary REPresentatie library;
  - `QMC/`: low-discrepancy (quasi-Monte Carlo) number generators;
  - `SE/`: contains ply file format library + simplify program needed for mesh decimation in density esitmation.
  - `IMAGE`: Image output in PPM or TIFF format.
- Actual implementation of rendering algorithms:
  - `GALERKIN/`: Galerkin radiosity;
  - `MCRAD/`: Monte Carlo radiosity: Hierarchical WDRS, Random Walk Radiosity and Stochastic Ray Radiosity;
  - `RAYTRACING/`: ray-casting, classical and stochastic raytracing;
  - `DEST/`: Density Estimation (contributed by Olivié Ceulemans, UCL/KUL Belgium);
  - `HURA/`: The Ultimate Rendering Algorithm (Het Ultieme Rendering Algoritme): for the time being just a template that you can copy to add your own rendering algorithms to `RenderPark`.

  The rule is that subdirectories can use data structures and global variables from their parent directory, but not vice versa, except for the auxiliary libraries such as `GDT` and `POOLS`.

1.3 Adding a new rendering algorithm

- First decide what class your rendering algorithm belongs to: world-space or pixel-based. This determines what routines ("methods") you have to implement in order to hook up your algorithm in the `RenderPark` program and how it will fit in the user interface. Suppose you want to add a world-space algorithm (adding a pixel-based algorithm is similar);
- make a separate sub-directory for your algorithm and copy the (template) files in the `HURA` subdirectory. Change the name 'hura' to something of your own;
- add a pointer to your rendering algorithm handle to the table of supported algorithms in `radiance.c`. If you want to add a pixel-driven algorithm, update `raytracing.c`;
- change the Makefile in order to compile and link you code as well when you do 'make' in the main directory;
- of course, implement the methods defined in `radiance.h` or `raytracing.h`.

Chapter 2

Data structures

2.1 The scene data structure

Scenes are represented using the following data structures and global variables (declared in `scene.h`):

- A linear list of material descriptions:
  `MATERIALLIST *MaterialLib`
  The type `MATERIALLIST` represents a linear list of material descriptions `MATERIAL` and is declared in `materiallist.h`. The type `MATERIAL` is declared in `material.h`. See also below, §2.7.

- A linear list of geometries:
  `GEOMLIST *World`
  The type `GEOMLIST` (`geomlist.h`) is a linear list of `GEOM` structures (`geom_type.h`). A `GEOM` structure represents all information about a geometry (object), see §2.2.

- A couple of data structures derived from the above when reading in a new scene:
  - `PATCHLIST *Patches`: linear list of all patches making up together the whole scene;
  - `GEOM *ClusteredWorldGeom`: hierarchical scene representation with an automatically generated and most often better hierarchy of bounding boxes;
  - `GRID *WorldGrid`: the scene packed into a voxel grid for raytracing acceleration (`grid`).

Lists and other data structures such as binary trees, octrees, ... are derived (using macros, see e.g. `patchlist.h`) from a generic implementation in the `GDT` (Generic Data Types) library.

2.2 The `GEOM` type

There are basically two types of geometries in the program:

- `COMPOUND` objects (`compound.h`: an aggregate object: basically a list of constituting `GEOM`etries, see §2.3;

- `SURFACE`s (`surface.h`): a primitive objects: basically a list of `PATCH`es (`patch.h`) that approximate the surface of some object, see §2.4;

`GEOM` should be seen as a superclass of `COMPOUND` and `SURFACE`. The implementation is in C and not C++ for historical reasons. We consider reimplementing everything in C++, after redesigning the

whole scene representation data structure. This has quite low priority however because the current implementation fulfills our needs well and has proven to be pretty reliable.

The GEOM structure (geom_type.h) contains:

- A pointer to object-specific data:
  `void *obj`
  For compound objects, it is a pointer to a COMPOUND struct (compound.h), for a surface object, it is a pointer to a SURFACE struct (surface.h);
- A pointer to a GEOM_METHODS struct (geom_methods.h). This structure contains pointers to a set of functions (methods) for manipulating the object data. This "methods" have the same interface for all kind of objects, but, of course, the implementation is different for different types of objects. For compound objects, the methods are implemented in compound.c, for surfaces in surface.c. Some examples of methods: ray-object intersection testing, requesting a list of constituting geometries (for a compound object), ...
- A bounding box (type BOUNDINGBOX, see bounds.h).
- A pointer to data for the geometry that are specific for a given rendering algorithm. When you need to associate specific data with a geometry in your rendering algorithm, you allocate space for this data and fill in this pointer for each geometry in the scene when initialising your algorithm. When terminating, you need to dispose of the allocated storage and set the pointer to NULL again. The pointer is initialised to NULL when reading in a scene. (see also radiance.h or the example in the HURA/ sub-directory).

2.3 COMPOUNDs

The object specific data for a compound object is nothing else than a pointer to a linear list containing the constituting GEOMetries. These constituting geometries may be surfaces or again compound objects, so you can build a hierarchical representation of your scene.

2.4 SURFACEs

Besides an identification number, useful for debugging, a SURFACE struct contains:

- a `MATERIAL *material` pointer to the material description of the object;
- a linear list `VECTORLIST *points` of coordinates of patch vertices;
- a linear list `VECTORLIST *normals` of vertex normal vectors;
- a linear list `VERTEXLIST *vertices` of vertex descriptions (see §2.5);
- a linear list `PATCHLIST *faces` of the patches that together make up the surface (see §2.6).

2.5 Vertices: the VERTEX type

For each vertex of a patch, the following information is kept (vertex.h):

- a pointer `VECTOR *point` to the vertex coordinates. Coordinates can be shared by several vertices. We store the coordinates only once for saving storage. It is also more efficient to compare a pointer to the coordinates than to compare the coordinates themselves if you want to know if two vertices have the same coordinates;
- a pointer `VECTOR *normal` to the vertex normal if specified in the input file, or `NULL` if no vertex normal was specified;
- a linear list `PATCHLIST *patches` of patches sharing the vertex. Useful for computing a vertex color as the average color of the patches sharing the vertex for smooth shading e.g.;
- a color (`RGB` type) to render the vertex when using Gouraud interpolation. This color needs to be computed and filled in the `VERTEX` structures by your rendering algorithm if you want to use the existing routines for smooth shaded rendering with graphics hardware;
- a pointer to rendering algorithm specific data attached to the vertex. This pointer is initialised to `NULL` when reading a scene and needs to be set to `NULL` again after you dispose of the data you attached to the vertex when your rendering algorithm is terminated. It is used in the same way as rendering algorithm specific data for `GEO`metries.

2.6 Faces: the `PATCH` type

This very crucial data type, declared in `patch_type.h`, contains besides some less important things:

- an ID-number for debugging and ID-rendering;
- an array `VERTEX *vertex[PATCHMAXVERTICES]` of `PATCHMAXVERTICES` (=4) pointers to the vertices of the patch (last pointer is a `NULL` pointer for triangles);
- the number of vertices (3 or 4);
- a back-pointer `SURFACE *surface` to the surface to which the patch belongs. Useful to find the material characteristics e.g.;
- a bounding box: this is a `NULL` pointer until a bounding box is needed the first time;
- patch plane normal and plane constant, midpoint, index of dominant normal component, jacobian of parameter transform, ...
- a color `RGB color`: to be filled in by your rendering algorithm in order to use graphics hardware for rendering your results;
- a pointer to rendering algorithm specific data for the patch, used in the same way as vertex or geometry specific data `radiance.h`;
- ...

2.7 Materials: the `MATERIAL` type

Besides a name for the material and an indication whether or not to consider surfaces of the given material as two-sided or one-sided surface, the `MATERIAL` type contains pointers to two functions that model the optical characteristics of the surface:

- **Emittance Distribution Function**: only defined for light sources. Determines how intense light is emitted by a light source as a function of direction;
- **Bidirectional Scattering Distribution Function**: describes how light is scattered (reflected or broken) at a given surface.

The corresponding data types `EDF` (`edf.h`) and `BSDF` (`bsdf.h`) are base classes in the same way as `GEOM` is a base class for `COMPOUND` and `SURFACE`. These base class structs basically contain a pointer to instance data and to methods to operate on the data.

2.7.1 Material data

The EDF/BSDF representation depends on the input file format (MGF or VRML).

MGF supports only a diffuse EDF and Phong-like reflection and refraction. The BSDF consists of separate data for reflection and refraction (`splitbsdf.[ch]`):

- *Bidirectional Reflectance Distribution Function*: describes how intense light coming from a first direction is being reflected into a second direction. Only defined for reflective materials;
- *Bidirectional Transmittance Distribution Function*: describes how intense incident light from a given first direction is being transmitted into a second direction (pointing into the other side of the material). Only defined for transparent materials;

Also `BRDF` (`brdf.h`) and `BTDF` (`btdf.h`) are base classes, similar to `BSDF` .... The implementation of the Phong-like BR/TDF can be found in `phong.c`.

VRML'97 is being extended with a very flexible and general EDF and BSDF model, extending the simple OpenGL-like material model that is available in the standard. The representation of this material model is not a part of RenderPark, but is kept in a separate library. The interface to this library can be found in `readvrml.C` and `PhBRML.C`.

2.7.2 Material methods

Besides routines for creating, printing, duplicating and destroying material data, the main EDF methods are:

- *Evaluation* of the EDF for a given point in space and (outgoing) direction;
- *Sampling* of a outgoing direction at a given point. Besides the sampled direction, also the probability density of generating the direction is returned.
- Computation of *emittance*: the integral of EDF times cosine w.r.t. the surface normal over the full hemisphere at a given point on a surface in the scene.

A distinction is made between diffuse, directional diffuse (glossy) and specular emission.

The main BSDF methods are:

- *Evaluation* of the BSDF at a given point of a surface of the scene and for given incoming and outgoing directions;
- *Sampling* of a outgoing direction for given incoming direction at a given point. Besides the sampled outgoing direction, also the probability density of generating the direction is required;
- Computation of the *albedo*: the integral over all outgoing directions of the BSDF for given incoming direction at a given point;

- Computation of the *refraction index* at a given point on a transparant surface.

In the former three routines, a restriction can be made to any combination of

- *reflection* or *refraction*;
- *diffuse* or *directional diffuse (glossy)* or *specular* component.

A restriction to only emission, reflection or refraction in any of the three glossiness ranges is needed for multipass methods. A radiosity algorithm will compute only diffuse illumination components. A second pass raytracing method will add the glossy and specular component. The "glossy" component can be computed and stored with radiosity-like algorithms but the border between glossy and specular is furthermore arbitrary. (Finite-element methods for directional diffuse environments have not yet been implemented in RenderPark.)

The implementation of these methods is in `phong.c` for the modified Phong model in MGF, and in the VRML library for VRML'97. The interface to the VRML material library is in `PhBRML.C`.

Chapter 3

Radiance methods

This section describes the actual implementation in RenderPark of world-space based rendering algorithms.

3.1 The radiance methods interface

All world-space based rendering algorithms provide an implementation of the interface defined in `radiance.h`. The main elements of this interface are:

- name of the world-space based rendering algorithm and other data for user interface buttons and command line options;
- a routine that initialises global variables of the algorithm to a default value. This routine is called once during initialisation of RenderPark, before loading scenes and before the user interface is created;
- a routine to parse and to print command line options for the algorithm;
- a routine that initialises the algorithm whenever a new scene is loaded or when the user selects the rendering algorithm for an already loaded scene: for instance, initialisation of radiosity data for each patch in the scene;
- a routine to perform one step of the computations, for instance one iteration of a radiosity algorithm;
- a routine to terminate the computations with the algorithm on the given scene. This routine frees all memory that was explicitely allocated in the initialisation routine or during the computations;
- a routine returning the radiance emitted at a given point on a given patch and into a given direction (for multipass methods or ray-casting of the result for instance);
- optional routines to allocate memory for algorithm-specific data to be associated with each patch in the scene, to print this data, and to free this memory. The allocation routine is called for every patch in the scene before the initialisation routine is called when loading a new scene, or selecting the algorithm for rendering an already loaded scene. These routines are not required: the same work could be done by the initialisation/termination routines, but they are sometimes convenient;
- routines for creating, destroying and displaying a control panel for the rendering algorithm, in order to let the user set options and parameters of the algorithm;

- a routine that shows a panel with statistics information about a run of the algorithm, for instance computation time, memory usage, ...;
- a optional routine to render the scene if the default hardware-based rendering routine is not sufficient. If this routine is not implemented, default hardware-based rendering (`render.c`, `rendercommon.c`) is used. The default rendering routine expects that a display RGB colour is filled in for every patch and vertex in the scene. A radiosity method for instance will compute and fill in these colour values and use the default rendering routine for display of the result;
- a routine to recompute display RGB colours of vertices and patches after for instance a change of the gamma correction parameter in the monitor calibration panel;
- a routine that allows the rendering algorithm to react on a change of material characteristics, currently unused;
- an optional routine to save the result of the computations in a VRML'97 file. This routine is only needed if the default VRML'97 saving routine (`writevrml.c`) is not sufficient. In general, a rendering algorithm will need both its own display routine and VRML saving routine, or none of both.

Chapter 4

Raytracing methods

4.1 Adding a ray tracing method

struct RAYTRACINGMETHOD: (raytracing.h) Provide a number of functions and data for a new raytracing method.

- **Variables**
  - **char *shortName**: Short name for the raytracing method, for use as argument of raytracing command line option.
  - **int nameAbbrev**: how short can the short name be abbreviated? */
  - **char *fullName**: Full name of the raytracing method.
  - **char *buttonName**: Name for the button for activating the method in the raytracing method menu.
- **Initialisation & Termination**
  - **void (*Defaults)(void)**: Function for setting default values etc... Called only upon startup of RenderPark. So switching between methods does not revert to default settings.
  - **void (*ParseOptions)(int *argc, char **argv)**: Function for parsing method specific command line options.
  - **void (*PrintOptions)(FILE *fp)**: Function for printing method specific command line options
  - **void (*Initialize)(void)**: Initializes the current scene for raytracing computations. Called when a new scene is loaded or when selecting a particular raytracing algorithm.
  - **void (*Terminate)(void)**: Terminate raytracing computations. Free up any memory that is still allocated for the method. Called when another raytracing method becomes active or upon exit.
- **Computing an image**
  - void (*Raytrace)(ImageOutputHandle *ip): Raytrace the current scene as seen with the current camera. If 'ip' is not a NULL pointer, write the raytraced image using the image output handle pointerd to by 'ip'.
  - void (*InterruptRayTracing)(void): Interrupts raytracing as soon as possible. (Normally sets an 'interrupt' flag)

- **Image control**
  - **int (*Redisplay)(void)**: Redisplays last raytraced image. Returns FALSE if there is no previous raytraced image and TRUE there is.
  - **int (*SaveImage)(ImageOutputHandle *ip)**: Saves last raytraced image in the file describe dby the image output handle.
- **User interface**
  - **void (*CreateControlPanel)(void *parent_widget)**: Function for creating user interface panel for interactively setting method specific parameters parent widget is passed as a void *. The type is however 'Widget', but we don't want to include the Xt header files in interface-independent files.
  - **void (*ShowControlPanel)(void)**: Show user interface control panel. Called when 'Control' button in raytracing menu is selected.
  - **void (*HideControlPanel)(void)**: Hide user interface control panel

To add a method provide a new 'RAYTRACINGMETHOD' structure and all the needed functions and add the new method to 'raytracing.c'

4.2 Utility classes/function for ray tracing

4.2.1 Screen iterator (screeniterate.C/H)

Utility functions to iterate all pixels in the image. A callback function must be provided that is called for every pixel in the image. The iterate functions show results on screen, but do not store the image!

Current available functions:

void ScreenIterateSequential(SCREENITERATECALLBACK callback, void *data) Iterate pixels from top left to bottom right.

ScreenIterateProgressive(SCREENITERATECALLBACK callback, void *data) 'Progressive' evaluation of pixels.

4.2.2 Class CPathNode

A path node is one vertex in a path. CPathNode contains all necessary information needed for evaluation of paths. E.g. hit point, hit surface, normal, bsdf evaluation, incoming direction, depth, pointer to next and previous node, ...

4.2.3 Sampler Classes

Sampler classes are used for building and extending paths. The most important function in these classes is 'Sample' which fills in a new path node, possibly depending on a current and previous path node.

Function: `virtual bool Sample(CPathNode *prevNode, CPathNode *thisNode, CPathNode *newNode, double x_1, double x_2);`

All 'Sample' functions return a boolean value that indicates if the new node is filled in. 'x_1' and 'x_2' are random numbers from 0.0 to 1.0.

Files: eyesampler.C/H

There are a number of different samplers available, all with a different purpose

Eye sampler

Sample a point on the eye surface. Currently a pinhole camera model is used, so that this sampler fills in the pathnode with the eye point.

Files: eyesampler.C/H

Pixel sampler

Sample a random point on a pixel. Trace a ray through that point and find the nearest intersection point. The new path node is filled in with information from this intersection point.

Files: pixelsampler.C/H

Surface samplers

Surface samplers fill in the new path node with a point somewhere in the scene that is usually generated by sampling a direction and tracing a ray from the current path node in that direction.

Files: bsdfsampler.C/H, specularsampler.C/H

Light sampler

A light sampler samples a certain point on a light source. This is used for next event estimation (Stoch. Raytracing) and for generating starting point for light paths (Bidir. path tracing).

Files: lightsampler.C/H

Light direction sampler

The current path node should be on a light source. A direction is chosen an a ray traced. A new path node is filled in with the new hit point information.

Files: lightdirsampler.C/H

4.2.4 Class CScreenBuffer

Class for storing pixel radiance information. Method examples are 'add radiance/flux to a specified pixel', ' render the buffer to screen (converts radiance to rgb)', 'save buffer as ppm or high dynamic range tiff', ...

Files: screenbuffer.C/H

4.3 Example: Tracing of a path

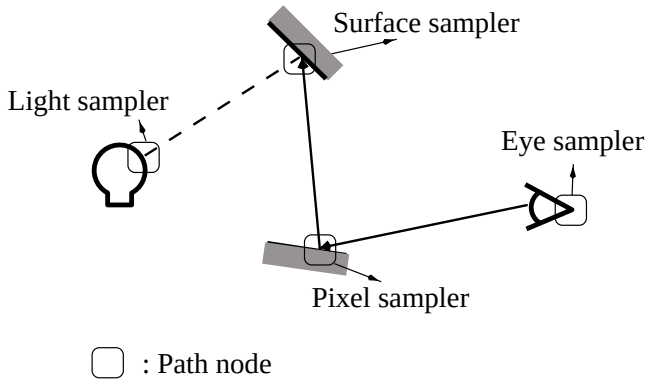

Figure 4.1: Example of how a path is traced and which samplers are used to create/fill in the respective path nodes