Application Development with BRL-CAD

Lee A. Butler
John Anderson
WARNING

Code Intensive Presentation

For Code Warriors Only!

non-programmers will need atropine, caffeine, and electro-shock therapy

(run, do not walk, to the nearest exit)
Overview

• Header files
• Shooting Rays
• Ray-Tracing User Interface Framework (RTUIF)
• Geometry Forms
• Creating Geometry
• Reading Geometry
• Modifying Geometry
# Header Files

- **The Big-6**

<table>
<thead>
<tr>
<th>Header</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>bu.h</td>
<td>libbu</td>
</tr>
<tr>
<td>bn.h</td>
<td>libbn</td>
</tr>
<tr>
<td>raytrace.h</td>
<td>librt</td>
</tr>
<tr>
<td>rtgeom.h</td>
<td>librt / libwdb</td>
</tr>
<tr>
<td>wdb.h</td>
<td>libwdb</td>
</tr>
<tr>
<td>vmath.h</td>
<td>(data types)</td>
</tr>
</tbody>
</table>
Prototype Application: rtexample.c

- Opens a database
- Retrieves geometry
- Prepares geometry for raytrace
- Performs raytrace

See source tree: rt/rtexample.c
Necessary Headers

```c
#include "conf.h"    /* compilation macros */
#include <stdio.h>
#include <math.h>
#include "machine.h" /* machine specific definitions */
#include "vmath.h"   /* vector math macros */
#include "raytrace.h" /* librt interface definitions */
```

- The “conf.h” and “machine.h” are ubiquitous in almost all BRLCAD apps
- The “raytrace.h” is present for geometry programs
  - Includes some additional headers
  - Contains most ray-tracing data structure definitions
Opening the Database

static struct rt_i *rtip; /* librt Instance structure */

/* rt_dirbuild() performs many functions for us */
rtip = rt_dirbuild(argv[1], buf, sizeof(buf));
if( rtip == RTI_NULL ) {
    fprintf(stderr,"rtexample: rt_dirbuild failure\n");
    exit(2);
}

• Opens database file
• Builds a “directory” of objects in the database
• Allows us to retrieve individual objects
if( rt_gettree(rtip, argv[2]) < 0 )
    fprintf(stderr,"rt_gettree(%s) FAILED\n", argv[2]);

• Retrieves tree top specified by argv[2] into a “working set” used by librt
rt_prep_parallel(rtip,1);

- Pre-computes useful terms for each primitive
  - Eg: triangle normals, function roots, trig terms
- Builds “space partition” tree to accelerate ra-trace
The application struct contains information about the ray that is to be computed and what should be done with the results.

```c
struct application ap;
ap.a_rt_i = rtip;
VSET( ap.a_ray.r_pt, 0, 0, 10000 );
VSET( ap.a_ray.r_dir, 0, 0, -1 );
ap.a_hit = hit;  /* where to go on a hit */
ap.a_miss = miss;  /* where to go on a miss */

(void)rt_shootray( &ap );  /* do it */
```
Application Struct

• Excerpts of application struct from raytrace.h:

```c
struct application {

    ...

    struct xray a_ray;    /* Actual ray to be shot */
    int (*a_hit)(struct application *,
                 struct partition *,
                 struct seg *);

    int (*a_miss)(struct application *);

    int a_onehit;              /* flag to stop on first hit */

    ...

    struct rt_i *a_rt_i;      /* this librt instance */

...}
```
Miss Routine

```c
miss( register struct application *ap)
{
    bu_log("missed\n");
    return(0); /* Value returned by rt_shootray() */
}
```

- Called when ray does not hit any geometry
Hit Routine

```c
hit(register struct application *ap, /* see raytrace.h */
    struct partition *PartHeadp) /* see raytrace.h */
{
    register struct partition *pp;
    register struct hit *hitp;
    point_t pt;
    for( pp=PartHeadp->pt_forw;
        pp != PartHeadp;
        pp = pp->pt_forw ) {
        hitp = pp->pt_inhit;
        VJOIN1( pt, ap->a_ray.r_pt, hitp->hit_dist, ap->a_ray.r_dir );
        VPRINT(“Hit Point”, pt);
    }
    return 1; /* value returned by rt_shootray();
}
```
Hit Routine Breakdown

```c
hit( register struct application *ap,
    struct partition *PartHeadp)
{
    register struct partition *pp;
    register struct hit *hitp;
    point_t pt;
    ...
```

- Partition Structure contains information about intervals of the ray which pass through geometry
- Hit structure contains information about an individual boundary/ray intersection
Partition Structure

struct partition {
    long pt_magic;           /* sanity check */
    struct partition *pt_forw; /* forwards link */
    struct partition *pt_back; /* backwards link */
    struct seg  *pt_inseg;   /* IN seg ptr (gives stp) */
    struct hit  *pt_inhit;   /* IN hit pointer */
    struct seg  *pt_outseg;  /* OUT seg pointer */
    struct hit  *pt_outhit;  /* OUT hit ptr */
    struct region *pt_regionp; /* ptr to containing region */
    char pt_inflip;          /* flip inhit->hit_normal */
    char pt_outflip;         /* flip outhit->hit_normal */
    struct region **pt_overlap_reg; /* NULL-terminated array of 
                                      * overlapping regions. 
                                      * NULL if no overlap. 
                                      */
    struct bu_ptbl pt_seglist;  /* all segs in this partition */
};

• From h/raytrace.h
Hit Structure

```c
struct hit {
    long    hit_magic;
    fastf_t hit_dist;    /* dist from r_pt to hit_point */
    point_t hit_point;   /* Intersection point */
    vect_t  hit_normal;  /* Surface Normal at hit_point */
    vect_t  hit_vpriv;   /* PRIVATE vector for xxx_() */
    genptr_t hit_private; /* PRIVATE handle for xxx_shot() */
    int     hit_surfno;  /* solid-specific surface indicator */
    struct xray *hit_rayp; /* pointer to defining ray */
};
```

• From raytrace.h
• Holds information about single ray/surface intersection.
  • Note: Only hit_dist filled in by librt.
hit(register struct application *ap, /* see raytrace.h */
  struct partition *PartHeadp) /* see raytrace.h */
{
  register struct partition *pp;
  register struct hit *hitp;
  point_t pt;
  for( pp = PartHeadp->pt_forw;
       pp != PartHeadp;
       pp = pp->pt_forw ) {
    hitp = pp->pt_inhit;
    VJOIN1( pt, ap->a_ray.r_pt, hitp->hit_dist, ap->a_ray.r_dir );
    VPRINT(“Hit Point”, pt);
  }
  return 1; /* value returned by rt_shootray();
}
Using the RTUIF

- Makes shooting grids of rays easy.
- Uses the same command line interface as \textit{rt}.
- Foundation for: \textit{rt}, \textit{rtweight}, \textit{rthide}, and other raytracing based applications.
- Simplest example shown in \textit{rt/viewdumy.c} in source tree
The 5 RTUIF Functions

- view_init
- view_setup
- view_2init
- view_pixel
- view_end
int view_init(struct application *ap, char *file, char *obj, int minus_o);

Called by main() at the start of a run. Returns 1 if framebuffer should be opened, else 0.

void view_setup(struct rt_i *rtip);

Called by do_prep(), just before rt_prep() is called, in “do.c”. This allows the lighting model to get set up for this frame, e.g., generate lights, associate materials routines, etc.

Void view_2init(struct application *ap);

Called at the beginning of a frame. Called by do_frame() just before raytracing starts.
int rayhit(struct application *ap, struct partition *PartHeadp);
\textit{Called via a\_hit linkage from \texttt{rt\_shootray}() when ray hits.}

int raymiss(struct application *ap);
\textit{Called via a\_miss linkage from \texttt{rt\_shootray}() when ray misses.}
void view_pixel(struct application *ap);

*Called by worker() after the end of processing for each pixel.*

void view_end(struct application *ap);

*Called in do_frame() at the end of a frame, just after raytracing completes.*
So Much for the Trivialities

• Now we look at actual geometry
Thinking About Geometry

• How to create it
• How to read it
• Doing anything useful with it
Geometric Representation

• BRL-CAD geometry has 3 forms:
  – External (Disk/DB)
    • Space efficient
    • Network integers (Big-Endian)
    • IEEE double-precision floating point (Big-Endian)
  – Internal (Editing)
    • Convenient parameter editing
    • Host float/int representation
  – Prep’ed (Raytrace)
    • Fast ray/primitive intersection
On-Disk Representation

- Space Efficient
- Machine independent
  - Only in new database format
- Database access is separate from object retrieval.
  - Database layer returns named objects.
    - Does not understand content.
  - Primitive objects get “Bag-o-Bytes” to turn into in-memory “internal” representation.
    - Have no knowledge of data origins
Internal Representation

• Convenient editing form
  – Host format floating point and integers
• Must be “exported” to be written to disk
• Primitive shape data structures defined in h/rtgeom.h
• Combination (and hence region) structure defined in raytrace.h
Prep’ed Representation

- The form that is actually raytraced
- Created from internal form by rt_prep() call
- May not include internal form
  - Saves memory
- May include additional fields
  - Precomputed values, additional data
Simple Database Application

- Necessary headers

```c
#include "conf.h"
#include <stdio.h>
#include "machine.h"
#include "vmath.h"
#include "raytrace.h"
#include "rtgeom.h"
#include "wdb.h"
```
Opening The Database

```c
struct rt_wdb *wdbp;
struct db_i *dbip = DBI_NULL;

/* open first, to avoid clobbering existing databases */
if ((dbip = db_open(argv[1], "r+w")) != DBI_NULL) {
    /* build a wdbp structure for convenient read/write */
    wdbp = wdb_dbopen(dbip, RT_WDB_TYPE_DB_DISK);

    if (db_dirbuild(dbip) < 0) {/* create directory database contents */
        bu_log("Error building directory for %s\n", argv[1]); exit(-1);
    }
} else {
    /* it doesn’t exist, so we create one */
    bu_log("doing wdb_fopen()\n");
    wdbp = wdb_fopen(argv[1]); /* force create */
}
```
Creating Geometry

• Note: All db units are in mm
  – Set mk_conv2mm global for other units

```c
point_t lo, hi;
...
/* add an axis-aligned ARB8 */
VSETALL(lo, 0.0);
VSETALL(hi, 2.0);
if (mk_rpp(wdbp, "mybox", lo, hi)) /* see libwdb for APIs */
    return -1;

/* add a sphere (really ellipse special case) */
if (mk_sph(wdbp, "myball", hi, 0.5)) /* see libwdb for APIs */
    return -1;
```
Getting Geometry

• To retrieve geometry, we have to get an internal representation

```c
struct rt_db_internal ip;
...
RT_INIT_DB_INTERNAL(&ip);
cond = rt_db_lookup_internal(wdbp->dbip, "mybox", &dp, &ip,
    LOOKUP_QUIET, &rt_uniresource);
If (!cond) { bu_log("couldn’t find %s
", “mybox”); exit(0);}
if (ip.idb_major_type == DB5_MAJORTYPE_BRLCAD /* see db5.h */ &&
    ip.idb_minor_type == ID_ARB8 /* see raytrace.h */ ) {

    struct rt_arb_internal *arb; /* see rtgeom.h */
arb = (struct rt_arb_internal *)ip.idb_ptr;
    RT_ARB_CK_MAGIC(arb);
    VPRINT("First Point", arb->pt[0]);
...
```
Primitive “Methods”

- Retrieved geometry has specific set of defined operations/methods available
- See h/raytrace.h for description of “struct rt_functab”
- Primitives should implement every method
  - Some do not. See librt/table.c for specifics
Putting Geometry Back

- Database I/O layer converts from internal to external format.

```
wdb_export(wdbp, "mybox", arb, ID_ARB8, mk_conv2mm);
```
Building Boolean Trees

- Regions/combinations used to store boolean trees.
  - Both are same type of database record
  - old “GIFT” form detailed here
- Simple boolean tree that contains
  - Names of objects
  - Boolean operations.
  - Matrix transformations
- Database record contains no actual geometry.
- Example code taken from
  - libwdb/wdb_example.c
Constructing Boolean List

• Build the list of elements first:

```c
struct wmember wm_hd; /* defined in wdb.h */
BU_LIST_INIT(&wm_hd.l);

/* see h/wdb.h or libwdb/reg.c for API conv/* or proc-db/* for examples */
(void)mk_addmember( "mybox", &wm_hd.l, NULL, WMOP_UNION );

/* If we wanted a transformation matrix for this element, we could have passed */
/* the matrix in to mk_addmember as an argument or we could add the following */
/* code: */
memcpy( wm_hd->wm_mat, trans_matrix, sizeof(mat_t));
/* Remember that values in the database are stored in millimeters, so the values */
/* in the matrix must take this into account. */

(void)mk_addmember("myball", &wm_hd.l, NULL, WMOP_SUBTRACT);
```
Regions/Combinations

• Constructing the actual combination record
  – Note: use mk_lcomb/mk_comb for initial creation only!
  • caveat: can use to update boolean tree under special conditions

```c
int is_region = 1;
VSET(rgb, 64, 180, 96); /* a nice green */

/* mk_lcomb is a macro using mk_comb.
* See libwdb/mk_comb() for full form */
mk_lcomb(wdbp,
    "box_n_ball.r", /* Name of the db element created */
    &wm_hd, /* list of elements & boolean operations */
    is_region, /* Flag: This is a region */
    "plastic", /* optical shader */
    "di=.8 sp=.2", /* shader parameters */
    rgb, /* item color */
    0); /* inherit (override) flag */
```
Retrieving A Combination

• Simple retrieval only gets:
  – List of elements
  – Boolean operations
  – Matrix transformations.

```c
struct rt_comb_internal *comb; /* see raytrace.h */
...
rt_db_lookup_internal(wdbp->dbip, "box_n_ball.r", &dp, &ip,
                      LOOKUP_QUIET, &rt_uniresource);

if (ip.idb_major_type != DB5_MAJORTYPE_BRLCAD /* see db5.h */ ||
    ip.idb_minor_type != ID_COMBINATION /* see raytrace.h */ ) {
    bu_bomb("gack\n");
}
comb = (struct rt_comb_internal *)ip.idb_ptr;
RT_CK_COMB(comb);
```
Combination Write-Back

- **Modify the boolean tree**
- **Write back out to db**

```c
/* Modify the combination we retrieved */
RT_GET_TREE(a, &rt_uniresource);
RT_GET_TREE(b, &rt_uniresource);

a->tr_l.tl_name = bu_strdup("newball");
a->tr_l.tl_op = OP_DB_LEAF;
a->tr_l.tl_mat = (matp_t)NULL;
a->tr_l.magic = RT_TREE_MAGIC;

b->tr_b.magic = RT_TREE_MAGIC;
b->tr_b.tb_left = comb->tree;
b->tr_b.tb_right = a;
b->tr_b.tb_op = OP_UNION;

comb->tree = b;
wdb_export(wdbp, "box_n_ball.r", comb, ID_COMBINATION, 1.0);
```
Combination Tree Info

• Need to “prep” the tree to obtain geometry
  – First, create “rt instance” struct rt_i object

```
struct rt_i *rtip; /* see raytrace.h */

/* if we’ve been doing db I/O */
rtip = rt_new_rti(wdbp->dbip);

/* if not already doing db I/O */
rtip=rt_dirbuild(filename, idbuf, sizeof(idbuf));
```
Processing combination tree

• Now to retrieve a treetop and prep:

```c
rt_gettree(rtip, "box_n_ball.r");
rtn_prep(rtip); /* now rtip has valid information */
```

• This could have been any level in the tree, not just a region.
Accessing Prepped Regions

- `rtip` has list of regions
- Access as a linked list
- Example: getting bounding box of regions…

```c
struct region *rp; /* see raytrace.h */

for (BU_LIST_FOR(rp, region, &rtip->HeadRegion)) {
    point_t tree_min, tree_max;
    VSETALL(tree_max, MAX_FASTF);
    VREVERSE(tree_min, tree_max);
    if (rt_bound_tree(rp->reg_treetop, tree_min, tree_max)) {
        bu_bomb("choke\n");
    }
    VPRINT("tree_min", tree_min); /* VPRINT is a macro from vmath.h*/
    VPRINT("tree_max", tree_max);
}
```
Making Temporary Changes

- Changes that only last for 1 application run
- Changes do not reside in on-disk database
Dynamic Geometry

• Involves special “inmem” database
  – Contains only modifications
  – Akin to “union” filesystem of Unix

• Directory structure tracks whether current version of object is on disk or in “inmem” database

• Object retrieval gets most current version

• Writes to inmem arranged though special wdb_dbopen() call
Accessing inmem database

- small difference in wdb_dbopen call
- all writes to this rt_wdb will go to “memory” database only

```c
struct rt_wdb *wdb_memp;
struct db_i *dbip = DBI_NULL;

if ((dbip = db_open(argv[1], "r+w")) != DBI_NULL) {
    /* The “INMEM” specifies that changes are to be made
       * ONLY in memory. Reads still come from disk for non-mem obj
       */
    wdb_memp = wdb_dbopen(dbip, RT_WDB_TYPE_DB_INMEM);

    if( db_dirbuild( dbip ) < 0 ) { /* create database content directory */
        bu_log( "Error building directory for %s\n", argv[1] ); exit(-1);
    }
}
```
Closing the Database

• Important to flush data and purge data structures!

  wdb_close(wdbp);
Thank you

Lee A. Butler
	butler@arl.army.mil
	410 278 9200