The CAD-EGS Project: Using CAD Geometries in EGS4[★]

WILLY G.J. LANGEVELD AND WALTER R. NELSON

*Stanford Linear Accelerator Center
Stanford University
Stanford, CA*

ABSTRACT

The objective of the CAD-EGS project is to provide a way to use a CAD system to create 3D geometries for use within EGS4. In this report, we describe an approach based on an intermediate file, written out by the CAD system, that is read by an EGS4 user code designed for the purpose. A prototype solution was implemented using a commonly used CAD system and the Virtual Reality Modeling Language (VRML) as an intermediate file format. We report results from the prototype, and discuss various problems arising from both the approach and the particular choices made.

★ Work supported in part by the Department of Energy contract DE-AC03-76SF00515

1. Introduction

1.1 GENERAL STRATEGY

The objective of the CAD-EGS project^[1] is to provide a way to use a CAD system to create 3D objects for use within EGS4^[2].

The most straightforward strategy for attaining the objective is to use an output file, produced by one of these CAD systems in some standard file format, as input to a special EGS user code^[3] designed for the purpose.

1.2 IMPLEMENTATION

The implementation presented here consists of the following parts:

1. A CAD program was chosen for demonstration purposes. Specifically, we chose MicroStation 95 by Bentley Systems, Inc. The reason for this choice is that MicroStation is already in use at SLAC.
2. An output file format was chosen. For various reasons, discussed in chapter 2, we chose VRML. MicroStation 95 supports this format with an add-on called MicroStation VRML Publisher. This choice is, however, not specific to MicroStation: many CAD systems support VRML in one way or another.
3. A user code for EGS4 was written that supports VRML input. The resulting program, called `cadegs`, is described in detail in chapter 3.

In chapter 4, we discuss issues that have not been dealt with in the current prototype, and possible future directions. In chapter 5, we present the results of benchmark calculations to demonstrate the accuracy and relative speed of the VRML approach compared to conventional geometry methods^[4] currently being used with EGS4. This paper ends with a summary of the conclusions.

2. The VRML file format

2.1 WHY VRML

In a previous memo^[5], we discussed the various 3D file formats that are used in the industry, and pointed out that there was no simple answer to the question of which file format should be used. We then concluded that the STEP^[6] and VRML^[7] file formats deserved further investigation.

After reviewing both of these file formats, we decided to base the prototype of the `cadegs` program on the VRML^[8] file format. This decision was based on number of considerations:

1. The STEP format is an ISO standard of great complexity. The RD44 collaboration plans to use it in Geant4^[9], the successor to the well-known event-generation and detector-simulation package Geant 3.21^[10]. See also Ref. 11. By comparison, the VRML standard is much simpler and easier to understand.
2. Source code for a VRML parser is available free of charge (although at the time of implementation only for VRML version 1.0 – see later). Free source code for STEP is only available in a preliminary form. For complete documentation and source code, a fee is required by the ISO.
3. Since the objective of the project was to demonstrate the feasibility of a version of EGS4 that can import geometry data from a CAD system, the standard it uses is of secondary concern, as long as the chosen standard can convey a sufficient amount of knowledge about the geometry. VRML is sufficient for the purpose of this demonstration, and a more complex standard can be adopted in a follow-on project, if desired.

2.2 VRML 1.0 vs. VRML 2.0

A complication arises from the fact that VRML is an evolving standard. The most current version of VRML is version 2.0, which is unfortunately (for our purposes) not backward compatible with version 1.0.

MicroStation 95, with its VRML Publisher add-on, supports and writes VRML 2.0-compliant output files. The current prototype of `cadegs` is based on a VRML 1.0 class library, and it cannot parse VRML 2.0 correctly. Since this is a technical detail, it does not invalidate the prototype solution presented here. A VRML 2.0 parser could easily be written and substituted in place of the current VRML 1.0 parser. For this project, however, we resort to minor edits to the VRML 2.0 file in order to make it VRML 1.0 compliant.

3. The `cadegs` program

3.1 OVERVIEW

The `cadegs` program consists of several parts:

1. A C translation of the EGS4 code system.
2. An EGS4 user code, `cadegs`, written in C++.
3. A C++ VRML parser, `QvLib`.
4. C++ 3D-vector and affine transformation classes.
5. A 3D display class and device independent plotting subsystem with several drivers, written in C++.

In the following sections we will discuss these components in some detail. After a short section on how to run `cadegs`, we conclude with a few preliminary results.

3.2 THE EGS4 CODE SYSTEM

While it is probably possible to link directly to the EGS4 object module, generated using the MORTTRAN preprocessor and the native Fortran compiler for any given system, some time can be saved by translating the Fortran source code to C first, using the `f2c` Fortran to C converter.

First, a sample EGS4 program was created in MORTTRAN. In the resulting Fortran code, the “main” program, `howfar()` and `ausgab()` user routines were then removed. The result was converted to C using `f2c`. In order to check that the resulting C code still works the same, the previously removed main program, `howfar()` and `ausgab()` were separately transformed into C using `f2c`, compiled using the native C compiler and then linked together with the C-compiled EGS4 code. The resulting `egs4job.out6` was compared with that from the Fortran version and found to be in agreement.

The object file of the C-version of the EGS4 code was subsequently used for the `cadegs` program. New C versions of the main program, `howfar()` and `ausgab()` routines were created, which call “user” routines `egs_main()`, `egs_howfar` and `egs_ausgab()`. These user routines are written in C++ and described in the next sections.

At this point, it should be noted that `f2c` was called with the following options:

```
f2c -A -r8 -C++ egs4.f
```

Here, the `-A` tells `f2c` to generate ANSI C code. The `-r8` options causes all `REALs` to be promoted to double precision. And finally, the `-C++` option causes C++ code to be produced (although the differences are minor). In order to get a particular version of EGS4 to behave correctly, some common blocks may have to be padded with dummy variables to ensure correct alignment for the double precision variables, and one or two other details may have to be fixed in the fortran source. In all cases the edits are minor.

Note, that the use of `f2c` to generate a C version of EGS4 also causes a few idiosyncrasies to appear. For one thing, certain library functions will be called that only exist in the GNU libraries (`f2c` is part of the GNU utilities). On UNIX systems it may therefore be necessary to link using the gcc front-end. For another, references to elements of an EGS4 common block appear as structure members. Particular attention needs to be paid to the fact that arrays in the translated code start at 0, while the translated common blocks also contain fields that index into these arrays. Such index variables assume, however, the Fortran standard of arrays starting at element 1 (although this would also be a problem when linking directly to Fortran code). While it is possible, and perhaps desirable, to write accessor functions for each of the common blocks and their elements which hide this from the user, in practice the number of places where one has to be aware of the problem is small, so the prototype does not go to that extreme.

3.2.1 The `cadegs/egs4` and `cadegs/cylegs4` directories

In the `egs4` directory, one finds, first of all, a subdirectory containing the original MORTRAN source code. This directory is called `mortransource`, and the contents should be familiar to EGS4 users and will not be discussed here.

There is also a `makefile` which creates the `egs4.o` and `egs4main.o` object files and the `egstest` test program which reproduces the results of the MORTRAN user code. The `makefile` takes the (slightly edited) Fortran source code `egs4.f` (without `main()`, `howfar()` and `ausgab()`) and turns it into `egs4.c` and then into `egs4.o`.

Some of the translated EGS4 common blocks are reflected in the header file `egs4.h` for convenient access by routines in other files. In addition, `egs4.h` declares prototypes for the accessible functions.

Interface routines to C++ versions of `main()`, `howfar()` and `ausgab()` are found in `egs4main.c`, which ultimately derives from `egs4main.f` but is changed significantly. It also contains several utility functions that facilitate dealing with EGS4 constructs.

The file `egstest.cpp` contains the “user” implementations of the routines `egs_main()`, `egs_howfar` and `egs_ausgab`, which in this case are C++ versions of the original MORTRAN test user code from the `mortransource` subdirectory.

And finally, this directory contains the usual `egs4job.dummy` and `egs4job.out6` generated by EGS4, and `egs4job.xsec` and (empty) `egs4job.in5` EGS4 input files.

The directory `cylegs4` is very similar to `egs4` (it contains a different version of EGS4) and does not need to be discussed separately.

3.3 THE `cadegs` USER CODE

The `cadegs` program is then, in essence, a set of three “user” routines, called `egs_main()`, `egs_howfar()` and `egs_ausgab()`. We will describe each of these in some detail for a sample application. The sample application reads a VRML file. Then, electrons are generated in the first octant of 3D space, and traced through whichever objects are encountered. The results are displayed in an adjustable perspective view, with an API similar to that of the EGS4 SHOWPL routine^[12]. The display code is based on the `p13` class which is discussed later. This `cadegs` program was also used in slightly modified form in order to perform consistency and timing benchmarks, the results of which are discussed in chapter 5.

In the following we first give an overview of the files in the `cadegs` and `cyltest` directories. We will then have a closer look at the source code for the user routines `egs_main()`, `egs_howfar()` and `egs_ausgab()`.

3.3.1 The `cadegs/cadegs` and `cadegs/cyltest` directories

The file `cadegs.cpp` contains the source for the three routines `egs_main()`, `egs_howfar()` and `egs_ausgab()`. When compiled and linked with everything else, it constitutes the program `cadegs`.

After reading in the VRML file, it traverses the object list created by the VRML parser, and collects a list of `CENode`'s—these are the CAD-EGS objects through which tracks will be traced. The actual VRML traverser code collecting the nodes is in `traverse.cpp`. Code implementing the list of nodes and the nodes themselves is found in `cestate*` and `cenode*`. The `*` here and in the following stands for three different file types. Files ending in `.cpp` contain source code implementing the functions that are not inline, files ending in `.h` contain the class declarations, files ending in `_il.h` contain source code implementing inline functions, and files ending in `.o` are object files generated by the compiler.

In order to be able to easily plot the results, a class called `MyCEState` is defined in `mycestate*`, which inherits from `pl3` and `CEState`. This class is instantiated in `egs_main()` and made globally available.

The equivalent of the EGS4 SHOWPL routine is implemented in `showpl*`.

The files `cadtest*` constitute a much simpler example program that does not include EGS4. It was used mainly for test purposes during the implementation of `cadegs`. The files `test.wrl` and `test2.wrl` are sample VRML files with simple geometries. There is finally the obligatory `makefile` and a file called `todo.txt` which contains a set of features/bugs that was or is yet to be implemented/fixd.

A similar set of files is present in the `cyltest` directory. These files are mostly identical to the ones in `cadegs`, with minor edits to generate the benchmark results discussed in chapter 5.

3.3.2 The `egs_main()` routine

In the following sections we will only discuss the sample `cadegs` program in the `cadegs` directory (the differences in the files in the `cyltest` directory are minor and self-explanatory).

The file `cadegs.cpp` starts with a bunch of include files, and is `ifdef`'ed for compilation on two platforms. The following includes are necessary for the VRML parser:

```
#include <QvDB.h>
#include <QvInput.h>
#include <QvNode.h>
#include <QvState.h>
```

Standard includes:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <math.h>
```

Next are includes for the packages `p13`, `egs4` and the `f2c` translator, the latter being needed by the `egs4` package:

```
#include <p13.h>
...
#include <f2c.h>
#include <egs4.h>
```

The following include files were described briefly before:

```
#include "cestate.h"
#include "mycestate.h"
#include "showpl.h"
```

Declaration of a setup function for EGS4, implemented later, and declaration of a pointer to a `MyCEState`, which is going to contain everything to be known about the `cadegs` state:

```
void egs_setup(void);
MyCEState *cestate = NULL;
```

The main `cadegs` program follows here:

```
void egs_main(int argc, char **argv)
{
    if (argc != 2) {
        printf("Usage: cadegs <input-filename>\n");
        return;
    }
}
```

Open the VRML file:

```
char *filename = argv[1];
```



```

FILE *fp = fopen(filename, "r");
if (fp == NULL) {
    printf("Unable to open file %s\n", filename);
    return;
}

```

Setup the QvLib VRML parser and read the file:

```

QvDB::init();

QvInput in;
QvNode *root;

in.setFilePointer(fp);

if (QvDB::read(&in, root) && root != NULL) {
    root->ref();
}
else {
    printf("Error while reading file %s\n", filename);
    fclose(fp);
    fp = NULL;
    return;
}

```

Get us a new MyCEState:

```

cestate = new MyCEState;

```

Now traverse the stored `QvNodes` and extract a list of CAD-EGS nodes (`CENodes`). In order to do this, we have added a `userdata` field to the `QvState` structure (see `include/QvState.h`—this is the main structure used by the VRML traverser in `traverse.cpp`), and we give it the pointer to our `CEState` structure. Inside the our traverser we then add `CENodes` to our `CEState` structure. For the details, see the `traverse.cpp` source code.

```

QvState state;

state.userdata = cestate;

```

```
root->traverse(&state);
root->unref();
```

We're now done with the VRML file, so we get rid of it:

```
fclose(fp);
fp = NULL;
```

We now have the list we need in `cestate`. Set up a plot device:

```
egs_showpl(1, 0);
```

All set. Initialize EGS4 (described later):

```
egs_setup();
```

Loop while there's no exit requested:

```
long replot = 0;
while (1) {
```

Check for user input

```
    replot = cestate->check_input();
```

The variable `replot` will be `-1` if the user requested to quit the program. Most of the time it will be `0`, i.e. the user didn't type anything, and in that case, it will just refresh the display and start another track. If `replot` is something other than `-1` or `0`, it means the user typed something, and we need to redisplay the configuration.

```
    switch(replot) {
        case -1 :
            goto cleanup;
            break;
        case 0 :
            egs_showpl(1, -1);
```

Now generate another track. This function is shown later—it eventually calls `egs_shower()`.

```
        cestate->track();
        break;
    default:
        egs_showpl(1, 0);
        break;
    }
}
```

When we get here, the user must have quit the program. Clean up the graphics device, and get rid of everything we created.

```
cleanup:
    egs_showpl(4, 0);
    if (cestate) delete cestate;
    return;
}
```

Below we list the code for `egs_setup()`. This routine was called from `egs_main()` above.

```
void egs_setup(void)
{
    long nreg;
    long i;
    long ixxst, jxxst;
```

First find the number of regions:

```
nreg = cestate->size();
```

Use one material (besides vacuum) for now, and make it iron:

```
egs_setmaterial(1, "fe");
```

Now fill MEDIA array. The medium for the first region is vacuum (material 0), all others are iron (material 1) for now:

```
egs_misc.med[0] = 0;
```

```
for (i = 1; i <= nreg; i++) egs_misc.med[i] = 1;
```

Set cutoffs for regions:

```
for (i = 0; i <= nreg; ++i) {
    egs_bounds.ecut[i] = 1.;
    egs_bounds.pcut[i] = 1.;
}
```

Note in the above, that the first element in the arrays, element 0, is what EGS4 (and `cadegs`) calls region 1. The shapes defined in the VRML file are numbered starting at 1, but are presented to EGS4 as regions with 1 added and thus start at number 2. Region 1 is therefore always reserved for “vacuum”. For the same reasons, the loops extend to and include `nreg`.

Now call HATCH:

```
egs_hatch();
```

Select starting random number seeds—unique pairs (`IXX`, `JXX`) define independent sequences:
 $1 \leq \text{IXX} \leq 31328$ and $1 \leq \text{JXX} \leq 30081$:

```
ixxst = 1;
jxxst = 1;
```

Starting seeds:

```
egs_random.ixx = ixxst;
egs_random.jxx = jxxst;
```

Initialize random number generator:

```
egs_rmarin();
return;
}
```

The next function sets up a track and calls the EGS4 shower routine:

```
long MyCEState::track(void)
{
```

```
double cost, sint, phi, cosp, sinp;  
double ei, ui, vi, xi, yi, zi, wi, wti;  
long iqi, iri;
```

The incident particle is an electron:

```
iqi = -1;
```

Incident total energy is 100 MeV:

```
ei = 100.0;
```

Coordinates of incident particle:

```
xi = yi = zi = 0.;
```

Now set up a direction vector in the first octant:

```
phi = drand48() * 0.5 * PI;  
sinp = sin(phi);  
cosp = sqrt(1.0 - sinp * sinp);  
  
cost = drand48();  
sint = sqrt(1.0 - cost * cost);  
  
ui = sint * cosp;  
vi = sint * sinp;  
wi = cost;
```

The incident particle starts out in region 1—vacuum:

```
iri = 1;
```

Weight factor—not used in this calculation, but is a parameter in subroutine SHOWER;
hence, define as unity:

```
wti = 1.;
```

And finally we call `egs_shower()`:

```
egs_shower(iqi, ei, xi, yi, zi, ui, vi, wi, iri, wti);
```

Since `egs_shower()` calls `egs_ausgab()` which in turn may plot some tracks, we should make sure at this point that all tracks are flushed to the graphics device:

```
PlotDevice *dev = cestate->getdevice();
if (dev) dev->flush();

return(0);
}
```

We will skip the description of the code that actually plots the VRML shapes through which particles are tracked. This code is also in `cadegs.cpp` and is rather easy to follow.

We will also not discuss the `egs_showpl()` routine here. The code for this function is in the file `showpl.cpp`.

3.3.3 The `egs_howfar()` routine

The “HOWFAR” routine is really the essence of the EGS4 user code. The CAD-EGS version is given below:

```
int egs_howfar(void)
{
    long i, ir;
    double deltaz;
```

Get the current stack pointer `i` and current region `ir`. Note the subtraction of 1 in both cases.

```
i = egs_stack.np - 1;
ir = egs_stack.ir[i] - 1;
```

Also get the vectors `p` and `q`.

```
vec p(egs_stack.x[i], egs_stack.y[i], egs_stack.z[i]);
vec q(egs_stack.u[i], egs_stack.v[i], egs_stack.w[i]);
```

Note that the EGS4 unit of length is cm, while the VRML unit of length is m:

```
p *= 0.01;
```

At this point, we could set DNEAR for future HOWFAR calls, (remembering cm/m conversion). We have however, not implemented this in the prototype:

```
/*
*   commented out:
    egs_stack.dnear[i] = 100.0 * cestate->nearestborder(p, &ir);
*/
```

Find the distance `deltaz` to the next item (see the discussion below):

```
deltaz = 100.0 * cestate->intersect(p, q, &ir);

if (deltaz < 0.0) {
    egs_epcont.idisc = 1;
    return(0);
}
```

Now check against USTEP and, if necessary, set the next region (adding 1).

```
if (deltaz <= egs_epcont.ustep) {
    egs_epcont.ustep = deltaz;
    egs_epcont.irnew = ir + 1;
}
return(0);
}
```

Clearly, all the work is done in the `cestate->intersect()` function. This routine takes the line $\mathbf{p} + \lambda \mathbf{q}$ and computes the intersection with the nearest boundary. The region `ir` we are currently in is a parameter given to this routine, and the next region is returned in the same variable `ir`. That is, if there is an intersection.

Specifically, the way it works is as follows. If we are currently in one of the shapes specified in the VRML file, it will compute the nearest exit point out of the shape along the specified line, without regard for any other shapes in its “world” (Clearly, this presents problems when one uses overlapping shapes. But since VRML does not support combinatorial geometry, there is no good solution for this problem. Further discussion about this can

be found in section 4.2). After computing the exit point, it returns “vacuum” (i.e. shape “0”, which turns into region 1) as the next shape—we cannot easily tell if there is a shape that adjoins the current shape exactly, so we might as well not try.

If we are currently in region 1 (vacuum), we loop over all the shapes in the “world” and determine the entry point along the specified line into the closest shape. In order to avoid a deadlock, extra logic is present in the intersection routines that uses the knowledge that we are entering or exiting and the direction of the normals to the surfaces of the shapes. When a closest entry point is found, the distance to it is returned and the `ir` variable is set to the shape being entered.

Some optimization has been done for these routines. In particular, recurring constants of the shape have been pre-calculated at “traversal” time, and bounding spheres, also calculated at traversal time, are used to avoid costly calculations for shapes the track will not pass through.

3.3.4 The `egs_ausgab()` routine

In the `cadegs` program, the `egs_ausgab()` routine is particularly simple:

```
void egs_ausgab(long iarg)
{
    egs_showpl(2, iarg);
    return;
}
```

It calls the `egs_showpl()` routine, in order to plot the tracks. In other cases, here is where one would score energy depositions etc. in the various objects. For example, the `cyltest` version of `egs_ausgab()` looks like this:

```
void egs_ausgab(long iarg)
{
    long i  = egs_stack.np - 1;
    long ir = egs_stack.ir[i] - 1;

    double edep = egs_epcont.edep;
    if ((edep > 0.) && (ir < 30) && (ir >= 0)) {
/*
*   Score deposited energy and number of tracks for regions
*/
        esum[ir] += edep;
        nsum[ir]++;
    }
```



```

    }
    return;
}

```

3.4 THE VRML PARSER

A public-domain VRML parser, written in C++ and called `QvLib`^[13], was used for this project.

The advantage, obviously, was that the code was available and free. The disadvantage, as mentioned before, is that it supports VRML 1.0 rather than VRML 2.0. The use of this pre-existing class library saved, however, a considerable amount of time and effort.

3.4.1 The `cadegs/include` directory

The header files for the `QvLib` classes are in the `cadegs/include` directory. Of these, only `QvState.h` was modified for use with `cadegs`: a field called `userdata` was added, to hold a pointer to the `MyCEState` class.

3.4.2 The `cadegs/src` directory

In the `cadegs/src` directory we find the implementation of the `QvLib` parser. The main file to be compiled is `QvLib.cpp` which basically includes all the other files. The `makefile` just compiles `QvLib.cpp` into `QvLib.o`.

For `cadegs`, one change was made to `QvLib.cpp`: the default traversal code, `QvTraverse.cpp`, is not included. This is because it is replaced by our own traversal code in the file `traverse.cpp` in the `cadegs/cadegs` directory.

3.4.3 The `cadegs/qvtest` directory

The `cadegs/qvtest` directory contains two test programs for `QvLib`: `readTest` and `travTest` with their respective source code and object files (and of course a `makefile`). There are also two sample VRML files (*.wrl).

Note that, in order to link these two programs, one has to undo the change in `QvLib.cpp` mentioned above and regenerate `QvLib.o`. This is because the two test programs require the default traversal code to be present.

3.5 THE AFFINE GEOMETRY CLASSES AND `p13`

For this project, a set of classes was written to facilitate vector operations in affine space. There is a `vec` class to represent 3D vectors, as well as a `vecset` class to handle sets of such vectors. Additionally, there is a `xfm` class which allows affine transformations to be applied to such vectors.

Another class was written to visualize 3D objects in a perspective view. This class is called `p13` and is built on top of the affine geometry classes above. It knows how to display (given a plot device, see next section) crude representations of various objects, such as lines, polygons, circles, spheres, cones and cylinders, in a perspective 3D view.

3.5.1 The `cadegs/p13` directory

The `vec` and `vecset` classes are contained in the files `vec*`, and the `p13` class in the files `p13*`. A class that handles single character unsolicited input for AIX can be found in `rawtty*`. The `makefile` compiles the sources and adds the object files to the library `libvecp13.a`. It also compiles two test programs, `vectest.cpp` and `p13test.cpp` which exercise the two sets of classes.

3.6 THE `plotpp` SUBSYSTEM, TEST FILES

The `p13` class described above sits on top of a “device independent” plotting package, called `plotpp`. It consists of an abstract `PlotDevice` class from which a number of specific plot drivers are derived. The currently available plot drivers are for the Amiga and for Tektronix 4105 (although Tektronix 4014 and PostScript drivers are also available with a small amount of effort).

3.6.1 The `cadegs/plotpp` directory

The `plotpp` package consists of the `PlotDevice` class in `plotpp*` and two drivers in `plotppamiga*` and `plotpptek4105*`. A few global utility functions for creating and deleting plot devices are in `plotdevices*`. The `makefile` compiles the sources and puts the object files in the library `libplotpp.a`.

3.7 HOW TO RUN `cadegs`

Since the default version of the `cadegs` program intends to plot the configuration of shapes and the tracks going through them, one currently needs a Tektronix graphics capable terminal in order to run `cadegs`. Of course, for a version of `cadegs` that does not plot anything, this restriction does not hold.

Before starting `cadegs`, you should set two environment variables. On AIX, you would do this as follows:

```
wglp09@morgan10 $ cd ~XXXXXX/cadegs/cadegs
wglp09@morgan10 $ setenv PLOTDEVICE tek4105
wglp09@morgan10 $ setenv egs4_path ../egs4
```

Here, XXXXXX stands for the location where you have installed the main `cadegs` directory. The first environment variable tells `cadegs` the plot driver you want, and the second one tells `cadegs` where to find the `egs4job.xsec` and `egs4job.in5` files. The directory you are in contains the `cadegs` executable, and `cadegs` expects write access to that directory, in order to output `egs4job.out6` and `egs4job.dummy` files.

Now, just run `cadegs` with a VRML input file as an argument:

```
wglp09@morgan10 $ cadegs test.wrl
```

Note that you may have to set up the terminal you are using (`xterm` for example) for use as a Tektronix terminal.

4. The VRML file format

4.1 WHERE TO GO FROM HERE

In the preceding, we have described in some detail the implementation of the prototype of `cadegs`. There are, however, a number of things that are not, or incorrectly, implemented, or that could be optimized. We briefly list them below.

4.1.1 VRML 2.0

The **cadegs** prototype VRML parser only understands files conforming to the VRML 1.0 specification. In order to work correctly with the output of MicroStation VRML Publisher—that is, without human intervention—it should of course understand VRML 2.0. Note, that VRML 2.0 also improves upon the types of shapes that can be used: VRML 2.0 provides a generalized **Extrusion** node, which can make the generation of complicated objects considerably easier.

4.1.2 The VRML LOD node

One particular feature of VRML is the Level Of Detail (**LOD**) node. This type of node is potentially very useful for Monte Carlo simulation programs. As the name implies, it allows the specification of more or less detail, depending on the distance from the object. Of course, one could also turn this around, and allow specification of the level of detail to use (*e.g.* on the command line) irrespective of the distance to the object, and allow faster or slower simulation in that fashion. In addition, the coarser detail could be used for estimating the “howfar” distance until the track is close to the object.

The prototype does not implement **LOD** correctly at this time.

4.1.3 Materials

While VRML allows the specification of a **Material** node, this can really not be used to specify the physical material (iron, aluminum, air) an object is made of. The properties it does allow one to specify are limited to those needed by ray-tracing software, such as **emissiveColor**, **specularColor**, **shininess** etc.

Of course one could pervert this node and agree upon a set of standards (red is iron, green is aluminum, etc.) but that seems rather ugly. The **cadegs** prototype does not adequately address this issue.

4.1.4 Optimizations

There is a number of optimizations that could be implemented in order to improve performance of the prototype. We just list a few of them here:

- The current implementation does not set the **dnear** value, but in many cases it could easily be computed.
- The implementation of the intersection routine for **IndexedFaceSet** nodes could use a better “point-inside-polygon” test. It currently uses a simple crossing test, that could easily be doubled in speed (using “bobbling”, for example).
- **IndexedFaceSets** could be stored in a manner such that it is easier to do the 2D projection before the crossing test.

4.2 VRML vs. STEP: LIMITATIONS OF VRML

The basic limitation that comes from the use of VRML is that it does not have enough expressive capabilities. Most importantly, it has no notion of “combinatorial geometry”. For example, one cannot use the `Cylinder` shape to generate a beam pipe. One would like to take two concentric `Cylinder` solids, and do a logical XOR (i.e. make the areas of overlap “empty”). VRML has no facilities for this. One is then tempted to ask if there is a file format (perhaps STEP?) that does have this capability. Our research into STEP did not extend far enough to know the answer. In VRML, the only way to approximate a shape like a beam pipe is to make an `IndexedFaceSet` that approximates the dual cylinder geometry in sufficient detail using polygons. The VRML 2.0 `Extrusion` node makes this fairly easy, but it, too, uses polygons.

Another limitation of VRML is the absence of any real knowledge of what an object is made of. The STEP standard does allow specification of the material, but there is no standardized way of doing that: for example, it is equally acceptable to specify “aluminum” as it is to specify “Al (purity 99.99%, extruded)”. Clearly, however, it is easier to specify a material in STEP.

VRML does have some constructs that would make it appear at first sight as if one can define “custom” shapes. It has a pair of operations, `DEF` and `USE`, which allow one to define a shape somewhere in the file and use it later. However, the `DEF` construct really only gives a name to a shape or group of shapes, and does nothing, for our purposes, that cannot be accomplished using other grouping nodes.

A more interesting construct (in VRML 2.0) is the `PROTO` construct. It allows one to build custom “prototype” shapes, with custom defined fields. Such fields must, however, be tied to fields in the description of the object, and the description of the object is in turn limited to pre-existing shapes and fields. Therefore there is nothing to be gained, for our purposes, from using `PROTO` constructs.

And finally, even if we could use such constructs to our benefit, it would still be an open question as to how one would convince a CAD program to generate them.

It should be noted here, that VRML is an evolving standard. This is clearly demonstrated by the changes from VRML 1.0 to VRML 2.0, and the fact that backward compatibility was abandoned in favor of a more consistent design. It is therefore not inconceivable that a follow-on project could include discussions with the VRML consortium^[14] to improve the language in the next revision—clearly it could benefit a lot from features such as those mentioned here.

Clearly STEP should be explored as an alternative. STEP is, however, a significantly complex system and it would add considerably to the size of any resulting program.

5. Benchmark comparison

The Crannell water-shower experiment^[15] has been one of the important benchmarks for testing all versions of the EGS Code System. In this experiment, a 1 GeV electron beam was incident on 8000 liters of distilled water in a stainless steel tank (122 x 122 x 460 cm³) and radial energy-deposition (MeV/cm³/electron) measurements were made at various water depths using a scintillation detector. EGS calculations have been shown to be in excellent agreement, on an absolute basis, with this experiment^[16].

As described earlier, the current limitation of the VRML approach is that it has no notion of “combinatorial geometry” which, in turn, prevents us from making a complete benchmark check against the Crannell experiment at this time. Nevertheless, we can test the **cadegs** user code to see if it accurately predicts the *longitudinal* development of the EM cascade. Therefore, without loss of generality, we have chosen as our test geometry a **single** cylinder of water having a radius of 12 cm, broken up into twenty slabs along the z-direction, each having a length of 20 cm. Figure 1 shows a screen shot of a sample plot generated by **cadegs** for this geometry.

Figure 1. A sample shower generated by **cadegs** in a VRML-based geometry. Tracks that leave the detector (see the description in chapter 5) are not plotted. Electrons with 1 GeV of kinetic energy enter from the far side.

The corresponding VRML file is quite simple:

```
#VRML V1.0 ascii
Separator {
#
# Set a viewpoint
#
    PerspectiveCamera {
        position 4.0 1.0 -1.0
        orientation 0 1 0 -.9
    }
#
# Transform so that cylinders are oriented parallel to Z axis
#
    Transform {
        rotation 1 0 0 1.570796326794897
    }
#
# And now a list of 20 cylinders
#
    Transform {translation 0 0 0.1}
    Cylinder {height 0.2 radius 0.12}
    Transform {translation 0 0 0.2}
    Cylinder {height 0.2 radius 0.12}

    ...

    Transform {translation 0 0 0.2}
    Cylinder {height 0.2 radius 0.12}
}
```

For comparison purposes, a standard SLAC user code called `ucgetrtz`, which treats cylinder-slab geometries using conventional `howfar()` techniques^{[3],[4]}, was run using the same geometry dimensions. In Figure 2 we have plotted the longitudinal energy-deposition fraction as a function of water depth. The runs were made on an IBM/RS-6000 with a PPC 604 processor at 200 MHz running AIX 4.1. For 10,000 cases, `cadegs` needed 283 seconds and `ucgetrtz` 254 seconds.

Figure 2. Longitudinal development of a 1-GeV shower in water: comparison of `cdegs` and `ucgetrtz`.

6. Conclusions

In this paper we have described a prototype design of a software system allowing the use of CAD geometries in EGS4. We showed that, while it is possible to take CAD geometries and use them inside EGS4, there are a number of limitations. Some of these are limitations of the particular approach chosen, and others may be limitations in general. We have shown that the prototype reproduces results produced by standard EGS4, without incurring a significant time penalty.

We conclude, based on what we have learned, that while the geometry problem is non-trivial, it can likely be solved given a sufficiently large amount of effort.

REFERENCES

1. Project Number 050, title “CAD-geometry interface for EGS4”, SLAC P.I. Walter R. Nelson.
2. “The EGS4 Code System”, Walter R. Nelson, Hideo Hirayama and David W.O. Rogers, SLAC Report 265 (1985).
3. “Writing SUBROUTINE HOWFAR for EGS4”, W.R. Nelson and T.M. Jenkins, SLAC TN-87-4 (31 August 1988 (Rev.)).
4. “Geometry Methods and Packages”, Chapter 17 in *Monte Carlo Transport of Electrons and Photons*, T.M. Jenkins, W.R. Nelson and A. Rindi (Editors), Plenum Press, 1988.
5. “CAD-EGS file formats and related issues”, W.G.J. Langeveld, SLAC internal memo (April 1996)
6. STEP is an ISO standard: ISO 10303.
7. VRML stands for Virtual Reality Modeling Language, and is based on the SGI OpenInventor format.
8. See for example: “The VRML 2.0 Handbook”, by Jed Hartman and Josie Wernecke, Silicon Graphics, Inc., Addison Wesley, 1996.
9. See <http://wwwinfo.cern.ch/asd/geant/geant4.html>
10. See <http://wwwinfo.cern.ch/asd/geant>
11. “Particle tracking in sophisticated CAD models for simulation purposes”, J. Sulkimo and J. Vuokoski, NIM A371 (1996) 434.
12. “Producing EGS4 Shower Displays With the Unified Graphics System”, Ray F. Cowan and W.R. Nelson, SLAC-TN-87-3 (August 1990 (Rev.)).
13. “QV: a quick VRML parser Version 1.0”, by Paul S. Strauss of Silicon Graphics.
14. See <http://www.vrml.org>
15. “Electron-Induced Cascade Showers in Water and Aluminum”, C.J. Crannell, H. Crannell, R.R. Whitney and H.D. Zeman, Phys. Rev. 184 (1969) 426.
16. “Solution of the Electromagnetic Cascade Shower Problem by Analog Monte Carlo Methods – EGS”, Chapter 12 in *Computer Techniques in Radiation Transport and Dosimetry*, W.R. Nelson and T.M. Jenkins (Editors), Plenum Press, 1980.