weinkauf/notes/amiramesh

Contents

Reading AmiraMesh Files for Scalar and Vector Fields

AmiraMesh is the native file format of **Amira**. The academic version of Amira is developed by the **Visualization and Data Analysis Group at Zuse Institute Berlin**. Commercial versions are available from **Visage Imaging, Berlin** and **VSG - Visualization Sciences Group, France**.

Some of the data sets on this website are given in AmiraMesh. This page explains briefly how to read them. While AmiraMesh is a very versatile file format for a large number of different data types, we concentrate here on the specifics required to load the data sets available from this website.

Overview of the File Format

An AmiraMesh file consists of a header and a data section. The header is always ASCII and contains the meta information such as grid type, bounding box, etc. The data section may be ASCII or binary. Here, we deal with binary data sections only. They are preceded by a line `# Data section follows` and another line `@1`.

```
<header, ASCII>
# Data section follows
@1
<data section, binary>
```

Header

The header of an AmiraMesh file looks as follows for a two-component vector field defined on a 3D uniform grid:

```
# AmiraMesh BINARY-LITTLE-ENDIAN 2.1 <== We have a binary (little-endian) data section.

define Lattice 4 6 8 <== Defines the uniform grid: number of grid points in x,y,z-direction.

Parameters {
    #~ BoundingBox: xmin, xmax, ymin, ymax, zmin, zmax
    BoundingBox -1 0 0 1 -0.5 0.5, <== Defines the bounding box of the uniform grid.
    CoordType "uniform" <== We have a uniform grid.
}

Lattice { float[2] Data } @1 <== The data section contains two floats for every grid point.

# Data section follows <== Precedes the data section.
@1
<binary data section>
```

For a scalar field, the lattice specification reads: `Lattice { float Data } @1`.

Data Section

The binary data section is written with these specifications:

- **Little-endian format:** This is the memory format of x86 processors and others. Hence, the data can be read into memory without any modifications on a PC or Intel-based Mac.
- **x-fastest:** To visit all grid points in the same order in which they are in memory, one writes three nested loops over the z,y,x-axes, where the loop over the x-axis is the innermost, and the loop over the z-axis the outermost.
- **Interleaved components:** The components u,v,w of a vector field are written interleaved, i.e., $[u_0, v_0, w_0]$, $[u_1, v_1, w_1]$, ..., where $[u_0, v_0, w_0]$ represents the first grid point, $[u_1, v_1, w_1]$ the second and so on.

Sample Code

Here we have a short C code to read AmiraMesh files as described above. You may use it as you wish, it is in the public domain. Note however, that you may want to use a better error handling (possibly exceptions), dynamic buffers, and so on. Also rest assured, that the routines in Amira itself look entirely different and are way more advanced and general. This here is a rather quick hack, but it gets the job done.

```
#include <stdio.h>
#include <string.h>
#include <assert.h>

/** Find a string in the given buffer and return a pointer
    to the contents directly behind the SearchString.
    If not found, return the buffer. A subsequent sscanf()
    will fail then, but at least we return a decent pointer.
*/
const char* FindAndJump(const char* buffer, const char* SearchString)
{
    const char* FoundLoc = strstr(buffer, SearchString);
    if (FoundLoc) return FoundLoc + strlen(SearchString);
```

weinkauf/notes/amiramesh

Contents

[vita]

[publications]

[gallery]

[teaching]

[notes]

[contact]

[links]

```
    that defines a scalar/vector field on a uniform grid.
*/
int main()
{
    //const char* FileName = "testscalar.am";
    const char* FileName = "testvector2c.am";
    //const char* FileName = "testvector3c.am";

    FILE* fp = fopen(FileName, "rb");
    if (!fp)
    {
        printf("Could not find %s\n", FileName);
        return 1;
    }

    printf("Reading %s\n", FileName);

    //We read the first 2k bytes into memory to parse the header.
    //The fixed buffer size looks a bit like a hack, and it is one, but it gets the job done.
    char buffer[2048];
    fread(buffer, sizeof(char), 2047, fp);
    buffer[2047] = '\0'; //The following string routines prefer null-terminated strings

    if (!strstr(buffer, "# AmiraMesh BINARY-LITTLE-ENDIAN 2.1"))
    {
        printf("Not a proper AmiraMesh file.\n");
        fclose(fp);
        return 1;
    }

    //Find the Lattice definition, i.e., the dimensions of the uniform grid
    int xDim(0), yDim(0), zDim(0);
    sscanf(FindAndJump(buffer, "define Lattice"), "%d %d %d", &xDim, &yDim, &zDim);
    printf("\tGrid Dimensions: %d %d %d\n", xDim, yDim, zDim);

    //Find the BoundingBox
    float xmin(1.0f), ymin(1.0f), zmin(1.0f);
    float xmax(-1.0f), ymax(-1.0f), zmax(-1.0f);
    sscanf(FindAndJump(buffer, "BoundingBox"), "%g %g %g %g %g %g", &xmin, &xmax, &ymin, &ymax, &zmin, &zmax);
    printf("\tBoundingBox in x-Direction: [%g ... %g]\n", xmin, xmax);
    printf("\tBoundingBox in y-Direction: [%g ... %g]\n", ymin, ymax);
    printf("\tBoundingBox in z-Direction: [%g ... %g]\n", zmin, zmax);

    //Is it a uniform grid? We need this only for the sanity check below.
    const bool bIsUniform = (strstr(buffer, "CoordType \"uniform\"") != NULL);
    printf("\tGridType: %s\n", bIsUniform ? "uniform" : "UNKNOWN");

    //Type of the field: scalar, vector
    int NumComponents(0);
    if (strstr(buffer, "Lattice { float Data }"))
    {
        //Scalar field
        NumComponents = 1;
    }
    else
    {
        //A field with more than one component, i.e., a vector field
        sscanf(FindAndJump(buffer, "Lattice { float["), "%d", &NumComponents);
    }
    printf("\tNumber of Components: %d\n", NumComponents);

    //Sanity check
    if (xDim <= 0 || yDim <= 0 || zDim <= 0
        || xmin > xmax || ymin > ymax || zmin > zmax
        || !bIsUniform || NumComponents <= 0)
    {
        printf("Something went wrong\n");
        fclose(fp);
        return 1;
    }

    //Find the beginning of the data section
    const long idxStartData = strstr(buffer, "# Data section follows") - buffer;
    if (idxStartData > 0)
    {
        //Set the file pointer to the beginning of "# Data section follows"
        fseek(fp, idxStartData, SEEK_SET);
        //Consume this line, which is "# Data section follows"
        fgets(buffer, 2047, fp);
        //Consume the next line, which is "@1"
        fgets(buffer, 2047, fp);

        //Read the data
        // - how much to read
        const size_t NumToRead = xDim * yDim * zDim * NumComponents;
        // - prepare memory; use malloc() if you're using pure C
        float* pData = new float[NumToRead];
```

weinkauf/notes/amiramesh

Contents

[vita]

[publications]

[gallery]

[teaching]

[notes]

[contact]

[links]

```
                if (NumToRead != ActRead)
                {
                    printf("Something went wrong while reading the binary data section.\nPremature end of file?\n");
                    delete[] pData;
                    fclose(fp);
                    return 1;
                }

                //Test: Print all data values
                //Note: Data runs x-fastest, i.e., the loop over the x-axis is the innermost
                printf("\nPrinting all values in the same order in which they are in memory:\n");
                int Idx(0);
                for(int k=0;k<zDim;k++)
                {
                    for(int j=0;j<yDim;j++)
                    {
                        for(int i=0;i<xDim;i++)
                        {
                            //Note: Random access to the value (of the first component) of the grid point (i,j,k):
                            // pData[((k * yDim + j) * xDim + i) * NumComponents]
                            assert(pData[((k * yDim + j) * xDim + i) * NumComponents] == pData[Idx * NumComponents]);

                            for(int c=0;c<NumComponents;c++)
                            {
                                printf("%g ", pData[Idx * NumComponents + c]);
                            }
                            printf("\n");
                            Idx++;
                        }
                    }
                }

                delete[] pData;
            }
        }

        fclose(fp);
        return 0;
    }
```

Test Data

The following files can be used to test an implementation.

- **testscalar.am**: a 3D scalar field
- **testvector2c.am**: a 3D vector field with two components at each grid point
- **testvector3c.am**: a 3D vector field with three components at each grid point

Running the code from above on testvector2c.am yields this **output**.