Discrete representations of geometric objects: Features, data structures and adequacy for dynamic simulation.

Part II: Levelsets & implicit surfaces
Motivation

✓ Accelerated geometric queries for problems such as:

➡ Is a point \((x^*, y^*)\) inside the object?

➡ Is a point \((x^*, y^*)\) within a distance of \(d^*\) from the object surface?

➡ What is the point on the surface which is closest to the query point \((x^*, y^*)\)?
Motivation

✓ Easy modeling of motions that involve topological change, e.g. shapes splitting or merging

✓ Such operations are difficult to encode with meshes, since they don’t “split” or “merge” unless we force them to
Implicit curves and surfaces (a.k.a. level-sets)

- Familiar representations address some of these demands:

  ✓ e.g. Analytic equations

    ➡ For an ellipse:

    \[
    \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1
    \]

    ➡ Easy inside/outside tests

    \[
    \frac{x_*^2}{a^2} + \frac{y_*^2}{b^2} < 1 \iff (x_*, y_*) \text{ is inside}
    \]
Implicit curves and surfaces (a.k.a. level-sets)

- Familiar representations address *some* of these demands:
  
  ✓ Describe a closed region via its boundary; split and reconnect when necessary

  ![Diagram](image)

  ➞ This may be tractable in isolated cases, but very cumbersome and impractical for more complicated cases, and with 3-dimensional surfaces
The level-set concept

- Represent a curve in 2D (or, a surface in 3D) as the zero isocontour of a (continuous) function, i.e.

\[ C = \{ (x, y) \in \mathbb{R}^2 : \phi(x, y) = 0 \} \]

e.g.

circle \( x^2 + y^2 = R^2 \equiv \{ (x, y) : \phi(x, y) = 0 \} \)

where \( \phi(x, y) = x^2 + y^2 - R^2 \)
The level-set concept

- This representation may seem redundant (we store information everywhere, just to capture a curve), but it conveys important benefits:
  
  ➡ Containment queries
  
  Is \((x_*, y_*)\) inside \(\mathcal{C}\)? \(\iff \phi(x_*, y_*) < 0\)

  ➡ Composability

  \[
  \begin{align*}
  \phi_1(x, y) &\text{ encodes } \Omega_1 \\
  \phi_2(x, y) &\text{ encodes } \Omega_2
  \end{align*}
  \]  
  \(\implies\)  
  \[
  \begin{align*}
  \max(\phi_1, \phi_2) &\text{ encodes } \Omega_1 \cap \Omega_2 \\
  \max(\phi_1, \phi_2) &\text{ encodes } \Omega_1 \cup \Omega_2
  \end{align*}
  \]

  ➡ We model both shape & topology change by simply varying the level set function
Levelset construction

- By the initial definition, there are many level set functions that encode the same shape

\[
\phi(x, y) = x^2 + y^2 - R^2 \\
\phi(x, y) = \sqrt{x^2 + y^2} - R \\
\phi(x, y) = e^{x^2+y^2} - e^{R^2}
\]

all encode the circle \( x^2 + y^2 = R^2 \)

- A specific systematic construction process: Signed distance functions

\[
\phi(x, y) < 0, \text{ if } (x, y) \text{ is inside } C \\
\phi(x, y) > 0, \text{ if } (x, y) \text{ is outside } C \\
\phi(x, y) = 0, \text{ if } (x, y) \text{ is on } C
\]

and \( |\phi(x, y)| = \text{distance of } (x, y) \text{ from } C \)
Levelset examples

\[ \phi(x, y) = \sqrt{x^2 + y^2} - R \]

\[ \phi(x, y) = \max\{x - R, -x - R, y - R, -r - R\} \]
Levelset properties

- We can offset the surface by a fixed distance $D$, by simply adding/subtracting $D$ from the levelset function.
- Proximity queries can be answered in $O(1)$ time.
  - e.g., is point $(x^*, y^*)$ within 0.1 units of the surface?
- The level set gradient is a unit normal, parallel to the direction of the closest point on the surface.
- We can project to the surface in $O(1)$ time:
  \[ (x_c, y_c) = (x, y) - \phi(x, y) \cdot \nabla\phi(x, y) \]
- SDFs are composable over unions/intersections of implicit domains.
Implementation
Implementation (adaptive)
• Lecture outline:
  • Obtaining the software
  • Set up and compilation
  • The debugging visualization tool
  • Setting up a basic, procedurally animated scene
  • Overview of basic data structures
Introduction to PhysBAM

- BSD licensed

- Two versions in existence
  - A fully featured version for modeling & dynamics
    - Used primarily for academic development
    - Not guaranteed to be stable
    - “Relatively” stable partial snapshots used in industry
  - A “reasonably” stable and tested open source version
    - Includes modeling tools, data structures, visualization and rendering
    - Excludes most dynamics algorithms (currently)
• Compatible (in principle) with several platforms
  • Has occasionally been built and deployed on Windows, Linux, MacOS, Android
  • Linux is considered the “primary” platform
  • Developers will currently support only Linux issues
• Recommended build system (Linux & Mac OS X): **SCons** *(use with MacPorts, on OS X)*

• Recommended compiler for public version: gcc 4.5.2
  • Versions >4.2.X should work as well.
  • Gcc 4.1.X has some known issues

• Makefiles can be used, but not supplied

• Can build on Windows with Cygwin/gcc

• Visual Studio mostly ok (certain template classes may pose issues, due to VC++ not being strictly standard-compliant).
Obtaining the code

• Developer website (Stanford)
  • http://physbam.stanford.edu (main page)
  • http://physbam.stanford.edu/links/download.html (download page)
  • Up-to-date with fixes & additions, but may change at any point in time

• CS838 snapshot (with some fixes and modifications):
  • http://pages.cs.wisc.edu/~cs838-2/software/PhysBAM-CS838.zip
  • Derived from public release (minus ray tracer)
Building PhysBAM (linux)

- Prerequisites: g++ (>4.2.X), scons, python (For visualization: freeglut, libjpg, libpng)

- Download and unzip the package (denoted by <basedir>, below)

- Set up PLATFORM environmental variable
  - On 64-bit systems, set to “nocona”
    - Using bash: “export PLATFORM=nocona”

- Set up symbolic links for SCons:
  - Go to <basedir>/Scripts/Archives/scons
  - Execute “python ./setup_scons.py”
• Try the compilation setup by building “opengl_3d” (the debugging visualization tool for 3D scenes)

• Go to the opengl_3d “Project” directory:
  <basedir>/Projects/opengl_3d

• Run SCons:
  • scons -u TYPE=release CXX=<C++ compiler> -j <# cores>
  • Substitute TYPE=debug to build in debug mode
  • Example:
    • scons -u TYPE=release CXX=g++44 -j 4

• Object/Library/Executable files placed in <basedir>/build

• A symbolic link “opengl_3d_nocona” is generated inside the project directory (or opengl_3d_nocona_debug, in debug mode)
The debugging visualization tool

- Separate executables for 1D, 2D, 3D
- Takes as argument the “output directory” produced by a simulation application
- 3D axes: Color coded R-G-B for x-y-z
- Left mouse: Rotate
- Middle mouse: Pan
- Right mouse: Zoom
- Most important key: “?” (interactive legend!)
• Two top-level directories

  • Public Library:
    Basic data structures and modeling object classes. Structured as a stand-alone library.
    (need not be modified for most tasks)

  • Projects:
    “User-space” applications, each in its own subdirectory. Structured such that each project is independent of others; should not cross-link.
• **Public_Library** structure

• Subdirectory **PhysBAM_Tools**:
  • Contains basic data structures, numerical solvers, and defines C++ classes that are not directly related to modeling and simulation (hashtables, arrays, graphs, matrices, vectors etc)
  • Each file is named after the SINGLE class it implements. Can search for class definitions by doing a tree search for the class name (with .h or .cpp)

• Subdirectory **PhysBAM_Geometry**:
  • Geometry data structures, related to physics simulation

• Subdirectory **PhysBAM_Rendering**:
  • Debugging visualizer + Ray tracer
Jump-starting a new project

• Create a new Project subdirectory
  • e.g. `<basedir>/Projects/physbam-test`

• Create a stub `SConscript` file, for compilation, e.g.
  ```python
  Import('env Automatic_Program')
  env=env.Copy(warnings_are_errors=0)
  Automatic_Program(env)
  ```

• Code in `main.cpp`, compile using SCons

• Run executable (“`physbam-test_nocona`”)

• Simulation and visualization and decoupled
  • Simulation outputs results to disk, without displaying them on-line
  • The debugging visualizer runs over the stored results, e.g.
    ```bash
    ../opengl_3d/opengl_3d_nocona <output_directory>
    ```
using namespace PhysBAM;

int main(int argc, char* argv[])
{
    typedef float T;
}
#include <PhysBAM_Tools/Log/LOG.h>
using namespace PhysBAM;

int main(int argc,char* argv[]) {
    typedef float T;

    LOG::Initialize_Logging();
    LOG::Finish_Logging();
}
```cpp
#include <PhysBAM_Tools/Log/LOG.h>
#include <PhysBAM_Geometry/Geometry_Particles/GEOMETRY_PARTICLES.h>
using namespace PhysBAM;

int main(int argc,char* argv[]) 
{
    typedef float T;

    typedef VECTOR<T,3> TV;

    LOG::Initialize_Logging();

    GEOMETRY_PARTICLES<TV> particles;

    LOG::Finish_Logging();
}
```
```cpp
#include <PhysBAM_Tools/Log/LOG.h>
#include <PhysBAM_Geometry/Geometry_Particles/GEOMETRY_PARTICLES.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/TRIANGULATED_SURFACE.h>
using namespace PhysBAM;

int main(int argc,char* argv[]) {
    typedef float T;

typedef VECTOR<T,3> TV;

LOG::Initialize_Logging();

GEOMETRY_PARTICLES<TV> particles;
TRIANGULATED_SURFACE<T>& cylinder_surface=*TRIANGULATED_SURFACE<T>::Create(particles);
cylinder_surface.Initialize_Cylinder_Mesh_And_Particles(20,20,5,1,false);

LOG::Finish_Logging();
}
```
```cpp
#include <PhysBAM_Tools/Log/LOG.h>
#include <PhysBAM_Geometry/Geometry_Particles/GEOMETRY_PARTICLES.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/TRIANGULATED_SURFACE.h>
using namespace PhysBAM;

int main(int argc,char* argv[]) {
    typedef float T;
    typedef VECTOR<T,3> TV;
    LOG::Initialize_Logging();

    GEOMETRY_PARTICLES<TV> particles;
    TRIANGULATED_SURFACE<T>& cylinder_surface=*TRIANGULATED_SURFACE<T>::Create(particles);
    cylinder_surface.Initialize_Cylinder_Mesh_And_Particles(20,20,5,1,false);

    LOG::Finish_Logging();
}
```
```cpp
#include <PhysBAM_Tools/Log/LOG.h>
#include <PhysBAM_Geometry/Geometry_Particles/GEOMETRY_PARTICLES.h>
#include <PhysBAM_Geometry/Solids_Geometry/DEFORMABLE_GEOMETRY_COLLECTION.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/TRIANGULATED_SURFACE.h>
using namespace PhysBAM;

int main(int argc,char* argv[]) {
    typedef float T;
    typedef VECTOR<T,3> TV;

    LOG::Initialize_Logging();

    GEOMETRY_PARTICLES<TV> particles;
    TRIANGULATED_SURFACE<T>& cylinder_surface=*TRIANGULATED_SURFACE<T>::Create(particles);
    cylinder_surface.Initialize_Cylinder_Mesh_And_Particles(20,20,5,1,false);

    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection(particles);
    collection.Add_Structure(&cylinder_surface);

    LOG::Finish_Logging();
}
```
```cpp
#include <PhysBAM_Tools/Log/LOG.h>
#include <PhysBAM_Tools/Read_Write/Utilities/FILE_UTILITIES.h>
#include <PhysBAM_Geometry/Geometry_Particles/GEOMETRY_PARTICLES.h>
#include <PhysBAM_Geometry/Solids_Geometry/DEFORMABLE_GEOMETRY_COLLECTION.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/TRIANGULATED_SURFACE.h>
using namespace PhysBAM;

int main(int argc,char* argv[]) {
    typedef float T;
    typedef float RW;
    RW rw=RW();STREAM_TYPE stream_type(rw);
    typedef VECTOR<T,3> TV;

    LOG::Initialize_Logging();

    GEOMETRY_PARTICLES<TV> particles;
    TRIANGULATED_SURFACE<T> &cylinder_surface=*TRIANGULATED_SURFACE<T>::Create(particles);
    cylinder_surface.Initialize_Cylinder_Mesh_And_Particles(20,20,5,1,false);

    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection(particles);
    collection.Add_Structure(&cylinder_surface);

    LOG::Finish_Logging();
}
```
```cpp
#include <PhysBAM_Tools/Log/LOG.h>
#include <PhysBAM_Tools/Read_Write/Utilities/FILE_UTILITIES.h>
#include <PhysBAM_Geometry/Geometry_Particles/GEOMETRY_PARTICLES.h>
#include <PhysBAM_Geometry/Geometry_Particles/REGISTER_GEOMETRY_READ_WRITE.h>
#include <PhysBAM_Geometry/Solids_Geometry/DEFORMABLE_GEOMETRY_COLLECTION.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/TRIANGULATED_SURFACE.h>
using namespace PhysBAM;

int main(int argc,char* argv[]) {
    typedef float T;
    typedef float RW;

    RW rw=RW();STREAM_TYPE stream_type(rw);
typedef VECTOR<T,3> TV;

    LOG::Initialize_Logging();

    Initialize_Geometry_Particle();Initialize_Read_Write_Structures();

    GEOMETRY_PARTICLES<TV> particles;
    TRIANGULATED_SURFACE<T>& cylinder_surface=*TRIANGULATED_SURFACE<T>::Create(particles);
cylinder_surface.Initialize_Cylinder_Mesh_And_Particles(20,20,5,1,false);

    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection(particles);
collection.Add_Structure(&cylinder_surface);

    LOG::Finish_Logging();
}
```
#include <PhysBAM_Tools/Log/LOG.h>
#include <PhysBAM_Tools/Read_Write/Utilities/FILE_UTILITIES.h>
#include <PhysBAM_Geometry/Geometry_Particles/GEOMETRY_PARTICLES.h>
#include <PhysBAM_Geometry/Geometry_Particles/REGISTER_GEOMETRY_READ_WRITE.h>
#include <PhysBAM_Geometry/Solids_Geometry/DEFORMABLE_GEOMETRY_COLLECTION.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/TRIANGULATED_SURFACE.h>
using namespace PhysBAM;

int main(int argc,char* argv[]) {
    typedef float T;
    typedef float RW;

    RW rw=RW();STREAM_TYPE stream_type(rw);
    typedef VECTOR<T,3> TV;

    LOG::Initialize_Logging();

    Initialize_Geometry_Particle();Initialize_Read_Write_Structures();

    GEOMETRY_PARTICLES<TV> particles;
    TRIANGULATED_SURFACE<T>& cylinder_surface=*TRIANGULATED_SURFACE<T>::Create(particles);
    cylinder_surface.Initialize_Cylinder_Mesh_And_Particles(20,20,5,1,false);

    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection(particles);
    collection.Add_Structure(&cylinder_surface);

    FILE_UTILITIES::Create_Directory("output/0");
    collection.Write(stream_type,"output",0,0,true);

    LOG::Finish_Logging();
}

Wednesday, September 12, 2012
#include <PhysBAM_Tools/Log/LOG.h>
#include <PhysBAM_Tools/Parsing/STRING_UTILITIES.h>
#include <PhysBAM_Tools/Read_Write/Utilities/FILE_UTILITIES.h>
#include <PhysBAM_Geometry/Geometry_Particles/GEOMETRY_PARTICLES.h>
#include <PhysBAM_Geometry/Geometry_Particles/REGISTER_GEOMETRY_READ_WRITE.h>
#include <PhysBAM_Geometry/Solids_Geometry/DEFORMABLE_GEOMETRY_COLLECTION.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/TRIANGULATED_SURFACE.h>
using namespace PhysBAM;

int main(int argc,char* argv[]) {
    typedef float T;
    typedef float RW;

    RW rw=RW();STREAM_TYPE stream_type(rw);
typedef VECTOR<T,3> TV;

    LOG::Initialize_Logging();

    Initialize_Geometry_Particle();Initialize_Read_Write_Structures();

    GEOMETRY_PARTICLES<TV> particles;
    TRIANGULATED_SURFACE<T>& cylinder_surface=*TRIANGULATED_SURFACE<T>::Create(particles);
cylinder_surface.Initialize_Cylinder_Mesh_And_Particles(20,20,5,1,false);

    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection(particles);
collection.Add_Structure(&cylinder_surface);

    for(int i=0;i<=10;i++){
        FILE_UTILITIES::Create_Directory("output/"+STRING_UTILITIES::Value_To_String(i));
collection.Write(stream_type,"output",i,0,true);

        for(int p=1;p<=particles.X.Size();p++)
    }

    LOG::Finish_Logging();
}
Basic data structures

- Arrays (dynamic)
  - Templatized as ARRAY<T>
  - Dynamically sizeable, with elbow room for amortized $O(1)$ insertion/resize cost.
  - One-based! (Unfortunate Fortran legacy)

- Vectors
  - Corresponding to “geometric” notion of vector, in the n-dimensional space
  - Fixed length (contrast with std::vector)
  - Doubly templatized as VECTOR<T,d>(e.g. VECTOR<float,3>)
Basic data structures

- Matrices (fixed size)
  - Templatized as $\text{MATRIX}<T,d>(\text{square})$ or $\text{MATRIX}<T,d_1,d_2>$
  - Coordinated templates and function with $\text{VECTOR}<T,d>$
  - Special cases: $\text{SYMMETRIC\_MATRIX}, \text{DIAGONAL\_MATRIX}, ...$
- Other structures
  - Arbitrary-size matrices and vectors (dense & sparse)
  - Hashtables, Graphs, Complex Numbers, Random Number Generators, etc.
  - Abstract numerical routines (Krylov methods, Newton...)
- More on Geometrical structures, in the next lectures