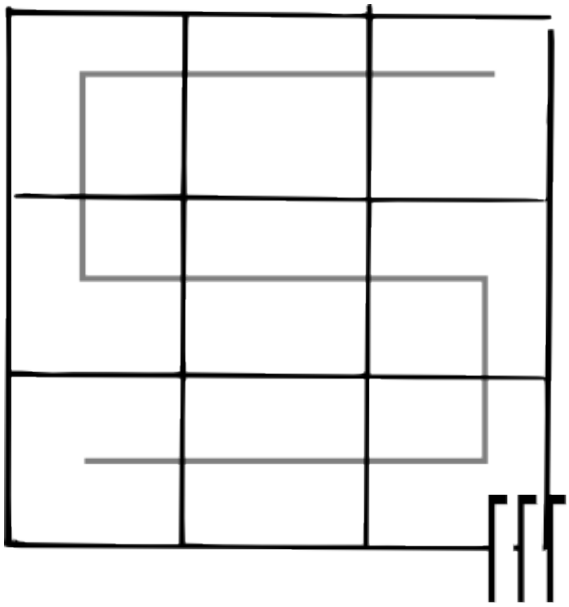


Peano Cookbook

www.peano-framework.org

Dr. rer. nat. Tobias Weinzierl

September 15, 2015

1 Preamble

Peano is an open source C++ solver framework. It is based upon the fact that spacetrees, a generalisation of the classical octree concept, yield a cascade of adaptive Cartesian grids. Consequently, any spacetree traversal is equivalent to an element-wise traversal of the hierarchy of the adaptive Cartesian grids. The software Peano realises such a grid traversal and storage algorithm, and it provides hook-in points for applications performing per-element, per-vertex, and so forth operations on the grid. It also provides interfaces for dynamic load balancing, sophisticated geometry representations, and other features. Some properties are enlisted below.

Peano is currently available in its third generation. The development of the original set of Peano codes started around 2002. 2005-2009, we merged these codes into one Peano kernel (2nd generation). In 2009, I started a complete reimplementation of the kernel with special emphasis on reusability, application-independent design and the support for rapid prototyping. This third generation of the code is subject of the present cookbook.

Dependencies and prerequisites

Peano is plain C++ code and depends only on MPI and Intel's TBB or OpenMP if you want to run it with distributed or shared memory support. There are no further dependencies or libraries required. C++ 11 is used. GCC 4.2 and Intel 12 should be sufficient to follow all examples presented in this document. If you intend to use Peano, we provide a small Java tool to facilitate rapid prototyping and to get rid of writing glue code. This Peano Development Toolkit (PDT) is pure Java and uses DaStGen. While we provide the PDT's sources, there's also a jar file available that comprises all required Java libraries and runs stand alone. To be able to use DaStGen—we use this tool frequently throughout the cookbook—you need a recently new Java version.

We recommend to use Peano in combination with

- Paraview (`www.paraview.org`) or VisIt (`https://wci.llnl.gov/simulation/computer-codes/visit/`) as our default toolboxes create vtk files.
- make and `awk`.

But these software tools are not mandatory.

The whole cookbook assumes that you use a Linux system. It all should work on Windows and Mac as well, but we haven't tested it in detail.

Who should read this document

This cookbook is written similar to a tutorial in a hands-on style. Therefore, it also contains lots of source code snippets. If you read through a chapter, you should immediately be able to re-program the presented details in your code and use the ideas.

Therefore, this cookbook is written for people that have a decent programming background as well as scientific computing knowledge. Some background in the particular application area's algorithms for some chapters also is required. If you read about the particle handling in Peano, e.g., the text requires you to know at least some basics such as linked-cell methods. The text does not discuss mathematical, numerical or algorithmic background. It is a cookbook after all.

What is contained in this document

This book covers a variety of problems I have tackled with Peano when I wrote scientific papers. There is no overall read thread through the document. I recommend to start reading some chapters and then jump into chapters that are of particular interest. Whenever something comes to my mind that should be added, I will add it. If you feel something is urgently missing and deserves a chapter or things remain unclear, please write me an email and I'll see whether I can provide some additional text or extend the cookbook.

September 15, 2015
Tobias Weinzierl

Contents

2 Quickstart

Time: Should take you around 15 minutes to get the code up and running. Then another 15 minutes to have the first static adaptive Cartesian grid.
Required: No previous knowledge, but some experience with the Linux command line and Paraview is advantageous.

2.1 Download and install

To start work with Peano, you need at least two things.

1. The Peano source code. Today, the source code consists of two important directories. The `peano` directory holds the actual Peano code. An additional `tarch` directory holds Peano's technical architecture.
2. The Peano Development Toolkit (PDT). The PDT is a small Java archive. It takes away the cumbersome work to write lots of glue code, i.e. empty interface implementations, default routines, ..., so we use it quite frequently.

For advanced features, you might want to use some **toolboxes**. A toolbox in Peano is a small collection of files that you store in a directory and adopt all pathes accordingly. From a user's point of view, when we use the term toolbox we actually mean this directory with all its content.

> **Remark:** Originally, we hoped that Peano's technical architecture (`tarch`) might become of value for several projects, i.e. projects appreciate that they do not have to re-develop things such as logging, writing of output files, writing support for OpenMP and TBB, and so forth. To the best of our knowledge, the tarch however is not really used by someone else, so we cannot really claim that it is independent of Peano. Nevertheless, we try to keep it separate and not to add anyting AMR or grid-specific to the tarch.

There are two ways to get hold of Peano's sources and tools. You either *download the archives from the website* or you *access the repository directly*. Both variants are fine. We recommend to access the respository directly.

2.1.1 Download the archives from the website

If you don't want to download Peano's whole archive, change to Peano's webpage `http://www.peano-framework.org` and grab the files

- `peano.tar.gz` and
- `pdt.jar`

from there. If you do so, please skip the first two lines from the script before. Otherwise, load down the important files with `wget`. Independent of which variant you follow, please unpack the `peano.tar.gz` archive. It holds all required C++ sources.

```
> wget http://sourceforge.net/projects/peano/files/peano.tar.gz
> wget http://sourceforge.net/projects/peano/files/pdt.jar
> tar -xzvf peano.tar.gz
```

There's a couple of helper files that we use IN the cookbook. They are not necessarily required for each Peano project, but for our examples here they are very useful. So, please create an additional directory `usrtemplates` and grap these files

```
> mkdir usrtemplates
> cd usrtemplates
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTKMultilevelGridVisualiserImplementation.template
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTKMultilevelGridVisualiserHeader.template
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTKGridVisualiserImplementation.template
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTKGridVisualiserHeader.template
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTK2dTreeVisualiserImplementation.template
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTK2dTreeVisualiserHeader.template
```

2.1.2 Access the repository directly

Instead of a manual download, you might also decide to download a copy of the whole Peano repository. This also has the advantage that you can do a simple `svn update` anytime later throughout your development to immediately obtain all kernel modifications.

```
> svn checkout http://svn.code.sf.net/p/peano/code/trunk peano
```

Your directory structure will be slightly different than in the example above, but this way you can be sure you grabbed everything that has been released for Peano through the webpage ever.

The archive `pdt.jar` will be contained in `pdt`, while the two source folders will be held by `src`. The directory `usrtemplates` is contained in `pdt`.

2.1.3 Prepare your own project

From hereon, we recommend that you do not make any changes within Peano repositories but use your own directory `peano-projects` for your own projects. We refer to one of these projects generically from hereon as `myproject`. Within `peano-projects`, we will need to access the directories `peano` and `tarch`. It is most convenient to create symbolic links to these files. Alternatively, you also might want to copy files around or adopt makefiles, scripts, and so forth. I'm too lazy to do so and rely on OS links.

```
> mkdir peano-projects
> cd peano-projects
> ln -s <mypath>/peano peano
> ln -s <mypath>/tarch tarch
> ls
```

```
  peano tarch
```

2.2 Create an empty Peano project

Peano projects require four files from the very beginning:

- A **specification** file is kind of the central point of contact. It defines which data models are used and which operations (algorithmic phases) do exist in your project. And it also specifies the project name, namespace, and so forth.
- A **vertex definition** file specifies which data is assigned to vertices in your grid.
- A **cell definition** file specifies which data is assigned to cells in your grid.
- A **state definition** file specifies which data is held in your solver globally.

We will use these files and modify them all the time. For our first step, they are basically empty. As mentioned before, we suggest to have one directory per project. Rather than creating the files as well as the directory manually, we can use the PDT for this:

```
> java -jar <mypath>/pdt.jar --create-project myproject myproject
> ls
  myproject peano tarch
```

If you are interested in the semantics of the magic arguments, call jar file without any argument and you will obtain a brief description. A quick check shows that the aforementioned four files now have been created:

```
> ls -al myproject
drwxr-xr-x 2 ... .
drwxr-xr-x 5 ... ..
-rw-r--r-- 1 ... Cell.def
-rw-r--r-- 1 ... project.peano-specification
-rw-r--r-- 1 ... State.def
-rw-r--r-- 1 ... Vertex.def
```

The PDT typically is used only once with the `--create-project` argument. From hereon, it serves different purposes. That is ...

2.3 A first spacetree code

... it helps us to write all the type of code parts that we don't want to write: **glue code** that does nothing besides gluing the different parts of Peano together.

We postpone a discussion of the content of the generated files to Chapter 3 and continue to run a first AMR example. For this, we call the PDT again. However, this time, we use the generated specification file as input and tell the tool to create all glue code.

```
> java -jar <mypath>/pdt.jar --generate-gluecode \
  myproject/project.peano-specification myproject \
  <mypath>/usrtemplates
```

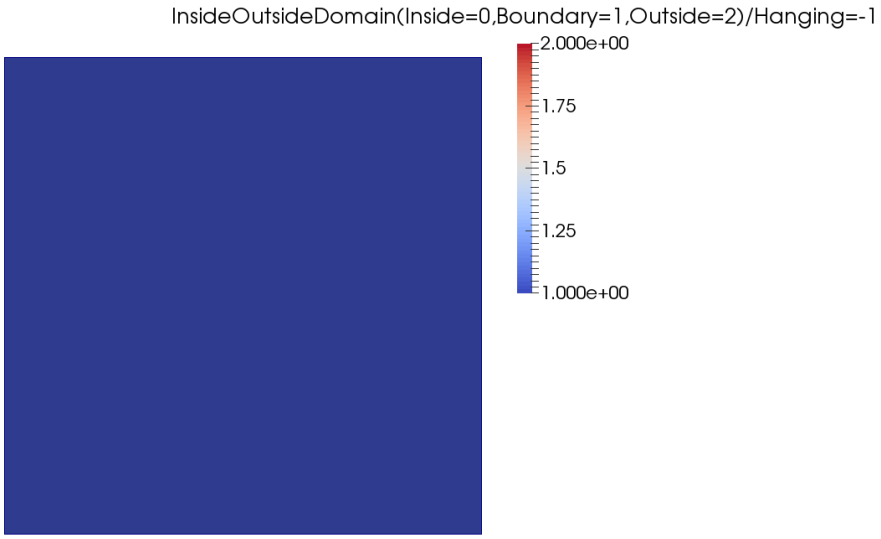
By default, the autogenerated, (almost) empty four files require the `usrtemplates`. We reiterate that many projects later won't need them. If we again study the content of our directory, we see that lots of files have been generated. For the time being, the `makefile` is subject of our interest. Depending on your compiler, you should be able to call `make` straight away. If it doesn't work, open your favourite text editor and adopt the makefile accordingly.

```
> ls myproject
  adapters Cell.cpp Cell.def
  Cell.h dastgen main.cpp
  makefile mappings project.peano-specification
  records repositories runners
  State.cpp State.def State.h
  tests Vertex.cpp Vertex.def
  Vertex.h VertexOperations.cpp VertexOperations.h
> make -f myproject/makefile
> ls
  files.mk myproject peano peano-YourProjectName-debug tarch
```

There it is: the first Peano executable. We can run it straight away:

```
> ./peano-YourProjectName-debug
> ls
  files.mk grid-0.vtk myproject peano
  peano-YourProjectName-debug tarch
```

We see that it has produced a vtk file. So it is time to startup Paraview or VisIt and see what is inside.



Congratulations: We have created the simplest adaptive Cartesian grid in 2d that does exist. A single square!

2.4 Some real AMR

We now set up something slightly more complicated. First of all, we switch to a 3d setup rather than 2d. For this, open the makefile (`myproject/makefile`) and alter the content of the `DIM` variable.

```
# Set Dimension
# -------------
#DIM=-DDim2
```

```
DIM=-DDim3
#DIM=-DDim4
```

If you clean your project (`make -f myproject/makefile clean`) and rebuild your code, you see that the individual files are translated with the compile switch

```
g++ ... -DDim3 ....
```

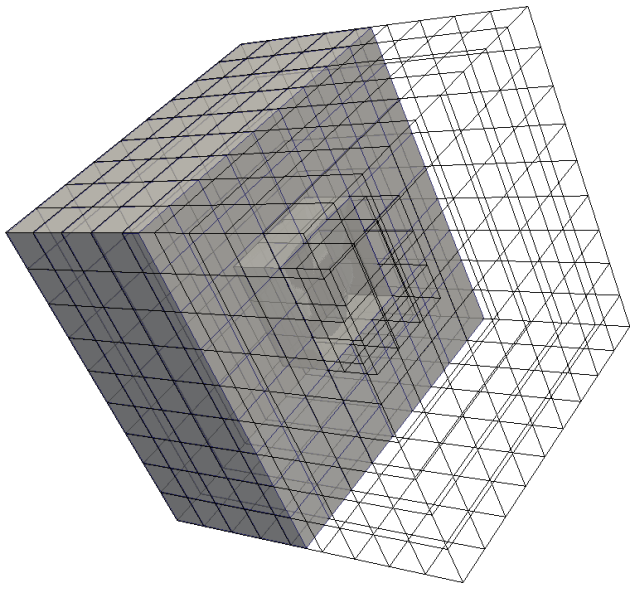
Indeed, this is all that's required for Peano to run a 3d experiment rather than a 2d setup.

> **Remark:** We do support currently up to 10-dimensional setups. If you require higher dimensions, you might even be able to extend Peano accordingly by changing solely the file `peano/utils/Dimensions.h`. But have fun with your memory requirements exploding.

Next, we will edit the file `myproject/mappings/CreateGrid.cpp`. Open it with your favourite text editor and search for the operation `createBoundaryVertex`. Change it into the code below:

```
void myproject::mappings::CreateGrid::createBoundaryVertex(
 myproject::Vertex& fineGridVertex,
 const tarch::la::Vector<DIMENSIONS,double>& fineGridX,
 const tarch::la::Vector<DIMENSIONS,double>& fineGridH,
 myproject::Vertex *const coarseGridVertices,
 const peano::grid::VertexEnumerator& coarseGridVerticesEnumerator,
 myproject::Cell& coarseGridCell,
 const tarch::la::Vector<DIMENSIONS,int>& fineGridPositionOfVertex
) {
  logTraceInWith6Arguments("createBoundaryVertex(...)", ...);
    // leave this first line as it is

  if (coarseGridVerticesEnumerator.getLevel()<2) {
    fineGridVertex.refine();
  }

  logTraceOutWith1Argument("createBoundaryVertex(...)",fineGridVertex);
}
```

If you compile this code and run the executable, you will (besides lots of debug output) obtain a way bigger vtk file. If you visualise it this time, we observe that the code refines towards the cube's boundary. You may want to play around with magic `2` in the operation above. Or you might want to continue to our final example.



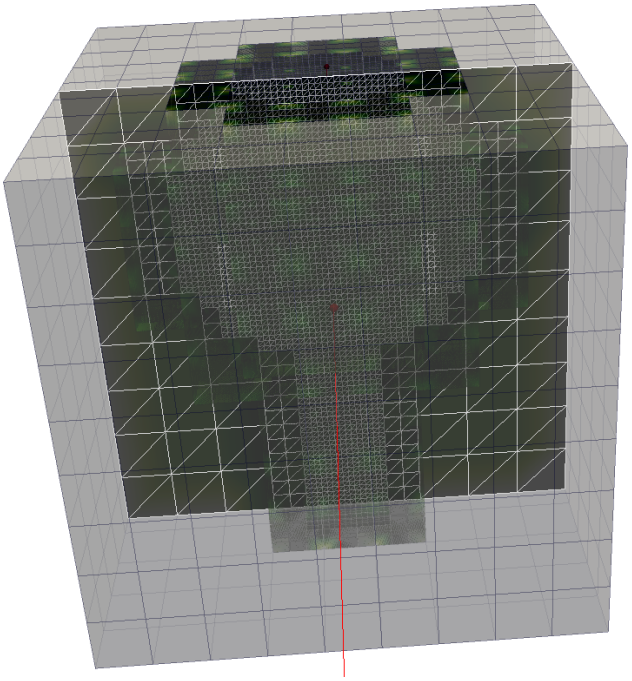
2.5 A tree within the spacetree

In the final example we create a slightly more interesting setup. We solely edit the operation `createInnerVertex` within the file `myproject/mappings/CreateGrid.cpp`, recompile it and have a look at the result. When you study source code, please note the similarity to Matlab when we work with vectors in Peano; as well as that the indices start with 0. If your want to get rid of all the debug statements and are sick of long waiting times, remove the `-DDebug` statement in the line `PROJECT_CFLAGS = -DDebug -DAsserts` within the makefile. There are more elegant ways to filter out log statements that we will discuss later.

```
void myproject::mappings::CreateGrid::createInnerVertex(
 myproject::Vertex& fineGridVertex,
 const tarch::la::Vector<DIMENSIONS,double>& fineGridX,
 const tarch::la::Vector<DIMENSIONS,double>& fineGridH,
 myproject::Vertex *const coarseGridVertices,
 const peano::grid::VertexEnumerator& coarseGridVerticesEnumerator,
 myproject::Cell& coarseGridCell,
 const tarch::la::Vector<DIMENSIONS,int>& fineGridPositionOfVertex
) {
 logTraceInWith6Arguments("createInnerVertex(...)",fineGridVertex,...);

 if (
   fineGridVertex.getRefinementControl()==Vertex::Records::Unrefined
   &&
   coarseGridVerticesEnumerator.getLevel()<4
 ) {
   bool trunk = (fineGridX(0)-0.5)*(fineGridX(0)-0.5)
              + (fineGridX(2)-0.5)*(fineGridX(2)-0.5)<0.008;
   bool treeTop = (fineGridX(0)-0.5)*(fineGridX(0)-0.5)
                + (fineGridX(1)-0.7)*(fineGridX(1)-0.7)
                + (fineGridX(2)-0.5)*(fineGridX(2)-0.5)<0.3*0.3;
   if (trunk | treeTop) {
     fineGridVertex.refine();
```

```
    }
  }

  logTraceOutWith1Argument("createInnerVertex(...)",fineGridVertex);
}
```

So here's what I get. Feel free to create better pics:



3 Basic Programming Course

Time: 15 minutes for the programming but perhaps around 30 minutes for the visualisation.
Required: Chapter 2.

In this section, we study a 2d example. Please adopt your makefile accordingly. Furthermore, we use the files `VTKMultilevelGridVisualiserHeader` and `...Implementation` as well as `VTK2dTreeVisualiser...`. If you have downloaded the whole Peano repository, these files can be found in `pdt/usrtemplates`. If not, you have to download them manually from the webpage. Please set up an empty project as discussed in Chapter 2 and implement one operation as follows (all other operations can remain empty/only filled with log statements):

```
void myproject::mappings::CreateGrid::touchVertexLastTime(
      myproject::Vertex& fineGridVertex,
      const tarch::la::Vector<DIMENSIONS,double>& fineGridX,
      const tarch::la::Vector<DIMENSIONS,double>& fineGridH,
      myproject::Vertex *const coarseGridVertices,
      const peano::grid::VertexEnumerator& coarseGridVerticesEnumerator,
      myproject::Cell& coarseGridCell,
      const tarch::la::Vector<DIMENSIONS,int>& fineGridPositionOfVertex
) {
  logTraceInWith6Arguments( "touchVertexFirstTime(...)", fineGridVertex, fineGridX, ...

  if (
    coarseGridVerticesEnumerator.getLevel()<5
    &&
    tarch::la::equals( fineGridX, 0.0 )
    &&
    fineGridVertex.getRefinementControl()==Vertex::Records::Unrefined
  ) {
    fineGridVertex.refine();
  }

  logTraceOutWith1Argument( "touchVertexFirstTime(...)", fineGridVertex );
}
```

This source fragment requires some additional explanation. We neglect the enumerator stuff for the time being. That will become clear later throughout the present chapter. The refinement control check says 'well, refine, but do it only on unrefined vertices'. It's just a matter of good style, not to call `refine` on a refined vertex. The middle line uses a function from the tarch's linear algebra namespace. It takes the `fineGridX` vector (the position of the vertex in space) and checks whether all entries equal zero. As we are working with floating point numbers, it is not a bit-wise check. Instead, it uses an interval of machine precision around zero. You may want to change this notion of machine precision in Peano (file `Scalar` within the `tarch::la` namespace). In general, it would be a good idea to study the content of the `la` component soon—there's lots of

useful stuff in there to work with tiny, dense vectors[1].

> **Remark:** Peano realises a **vertex-based**, **logical-or** refinement: You can invoke `refine` on any unrefined vertex. Peano then refines all cells around a vertex in the present or next traversal (it basically tries to do it asap, but sometimes data consistency constraints require it to postpone the actual refinement by one iteration). The other way round, you may read it as follows: A cell is refined if the refinement flag is set for any adjacent vertex.

Whenever you use Peano, you have to do three things:

1. Decide which algorithmic phases do exist and in which order they are called. Examples for algorithmic phases could be: set up grid, initialise all variables, refine regions of interest, perform an iterative solve step, plot some data, compute metrics on the solution, ...
2. Model the data, i.e. decide which data is assigned to the vertices and cells of the grid.
3. Implement the different actions on this data model that are used by the algorithmic phases.

This scheme lacks the bullet point 'run through the grid'. Indeed, Peano applications do never run themselves through the spacetree. They specify which set of operations is to be called throughout a run through the grid, i.e. they say what is done on which data. Afterward, they invoke the iteration and leave it to Peano to run through the grid and invoke these operations in the right order on the right ranks using all the cores you have on your machine[2]. This scheme realises something people call 'The Hollywood Principle':Don't call us, we call you!

> **Remark:** The **inversion of control** is the fundamental difference of Peano to other spacetree-based codes offered as a library. And typically it is the property many users first struggle with. Often, people claim 'I have to run through the grid this and that way'. Often, they are wrong. It can become quite comfortable to leave it to someone else to decide how grid traversals are realised. And it allows the grid traversal in turn to optimise the code under the hook without an application developer to bother.

3.1 On the power of loosing control

The algorithmic phases, i.e. what can be done on a grid, are specified in the specification file. Open your project's file. There are two different parts of the document that are of interest to us: An *event mapping* is an algorithmic step that you have to implement yourself. In this chapter's example, we want to do two things: create a grid and count all the vertices. Furthermore, we want to plot our grid, but let's keep in mind that Peano has some predefined actions as well. So we augment our mapping set as follows:

```
// Creates the grid
event-mapping:
  name: CreateGrid
```

[1] Peano someday should perhaps be rewritten to use boost linear algebra or some fancy template library. Feel free to do so. Right at the moment, it is all plain hand-craftet routines.

[2] This statements requires explanation, and indeed it is not *that* straightforward. But the idea is phrases correctly: the application codes specifies what is to be done and then outsources the scheduling and the responsibility to use a multicore machine to Peano.

```
// Counts all the vertices within the grid
event-mapping:
  name: CountVertices
```

Event mappings cannot be used directly. Instead, we have to specify adapters. Adapters take the tree traversal and invoke for each grid part a set of events. As we distinguish adapters which basically just glue together (multiple) events from the events themselves, we will be able to do the following later: we write a fancy visualisation routine, a routine that adopts the grid to a new data set and some compute routines. As we have done this in three different event sets, we can then combine these events in various ways: compute something and at the same time plot, compute only, plot and afterward adopt the grid, and so forth. For the time being, we use the following adapters:

```
adapter:
  name: CreateGrid
  merge-with-user-defined-mapping: CreateGrid

adapter:
  name: CountVertices
  merge-with-user-defined-mapping: CountVertices

adapter:
  name: CreateGridAndPlot
  merge-with-user-defined-mapping: CreateGrid
  merge-with-predefined-mapping: VTKGridVisualiser(finegrid)
  merge-with-predefined-mapping: VTKMultilevelGridVisualiser(grid)

adapter:
  name: CountVerticesAndPlot
  merge-with-user-defined-mapping: CountVertices
  merge-with-predefined-mapping: VTKMultilevelGridVisualiser(grid)

adapter:
  name: Plot
  merge-with-predefined-mapping: VTKGridVisualiser(finalgrid)
```

The first two adapters are trivial: They basically delegate to one event set. The next two take one event set each and invoke it. Furthermore, they also use a predefined event set. They will call `CreateGrid` or `CountVertices`, respectively, and at the same time plot. If you create all code with

```
java -jar <mypath>/pdt.jar --generate-gluecode
myproject/project.peano-specification myproject <mypath>/usrtemplates
```

it is the directory `usrtemplate` where the PDT searches for the predefined event sets. The last adapter by the way is a trivil one, too: It invokes only one of the events that ship with Peano.

Next, please create all glue code and have a quick look into the file `runners/Runner.cpp`. This file is the starting point of Peano. The C++ main routine does some setup steps and then creates an instance of the Runner (see the source code yourself if you don't believe). It then invokes `run()` which in turn continues to `runAsMaster` or `runAsWorker()`. The latter will play a role once we use MPI. For the time being, let's focus on the master's routine. Here, we see the following:

```
int myproject::runners::Runner::runAsMaster(myproject::repositories::Repository& repository) {
  peano::utils::UserInterface userInterface;
  userInterface.writeHeader();
```

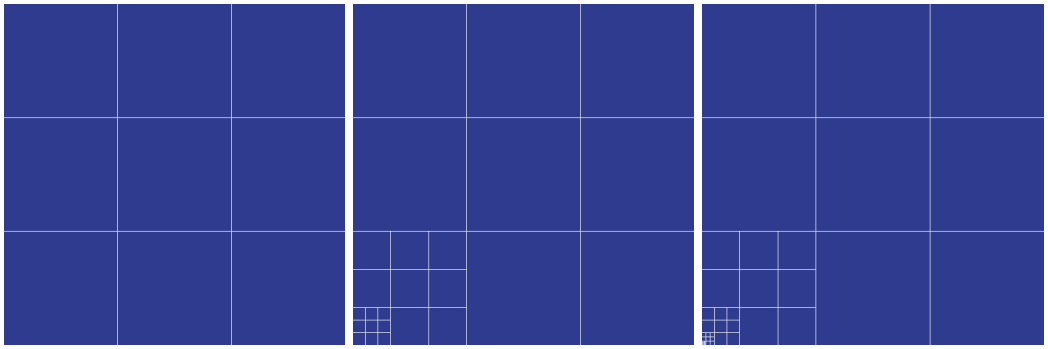
```
  // @todo Insert your code here

  // Start of dummy implementation

  repository.switchToCreateGrid(); repository.iterate();
  repository.switchToCountVertices(); repository.iterate();
  repository.switchToCreateGridAndPlot(); repository.iterate();
  repository.switchToCountVerticesAndPlot(); repository.iterate();
  repository.switchToPlot(); repository.iterate();



  repository.logIterationStatistics();
  repository.terminate();
  // End of dummy implementation

  return 0;
}
```

The PDT cannot know what exactly we do, so it basically runs all the adapters we have specified. This is the place where we implement our overall algorithm, i.e. the big picture. Lets change it as follows:

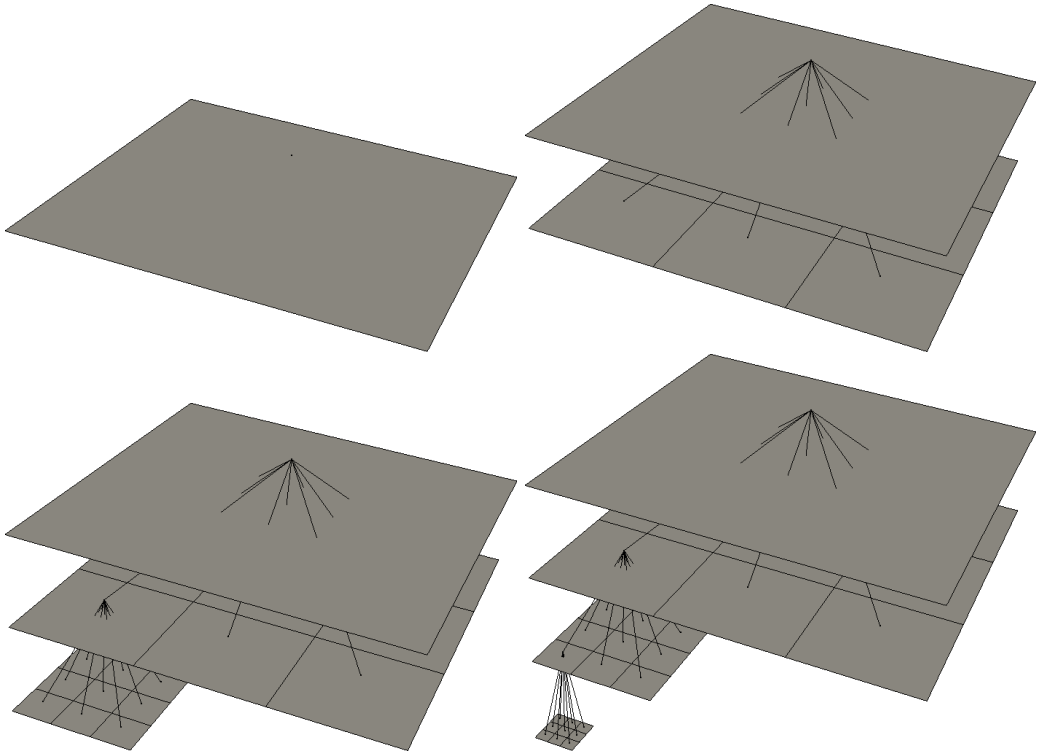
```
  peano::utils::UserInterface userInterface;
  userInterface.writeHeader();

  repository.switchToCreateGridAndPlot();
  for (int i=0; i<10; i++) repository.iterate();
  repository.switchToCountVertices(); repository.iterate();

  repository.logIterationStatistics();
  repository.terminate();

  return 0;
```

This algorithm says that we want to create a grid and at the same time plot it. We want to do this ten times in a row. Afterward, we switch to our vertex counting and want to run through the grid once more. This time, nothing shall be plotted. We just want to know how many vertices there are.

We really do not care about how the code runs through the grid. We also do not really care how all vertices, cells, whatever are processed. We say what is done in which order from a bird eye's perspective. If you compile the code and run it now, you should end up with a sequence of vtk files. You might want to make a video (take the files that are called `finegrid-`something). Below are some screenshots:



3.2 What happens



To understand what is happening, we can read all the outputs written to the terminal by Peano (see the next chapter how to remove them/filter them). Or we can just give a quick sketch:

1. We first create a repository. A repository is basically the spacetree, i.e. the grid, plus all the adapters we've defined. It also provides some statistics the can read out.
2. The code runs into the Runner's `runAsMaster()` routine and selects which adapter to use. This is the switch statement. In a parallel code, all involved ranks would now immediately active this adapter.
3. In the runner, we then call the `iterate` operation on the repository. The repository now starts to run through the whole grid. Actually, it runs through the spacetree in a kind of top-down way.
4. Whenever it encounters an interesting situation (it loads a vertex for the very first time, e.g.), it triggers an event. Event means that it calls an operation on the adapter.

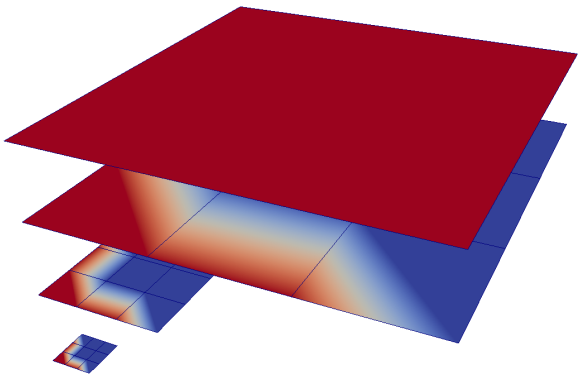
5. An adapter may fuse multiple events; both predefined and user-defined. Per event, it calls all these mappings' implementations one after another. The order is the same you have used in your specification file.

> **Remark:** This document does not run through the list of available events, in which order they are called and so forth. You may want to have a look into any event's header. The PDT augments the header with a quite verbose explanation what event is called when.

It might now be the right time to look into one of these mapping headers to get a first impression what is available. Basically, they describe all important plug-in points that you might want to use in any element-wise multiscale grid traversal.

3.3 Data model

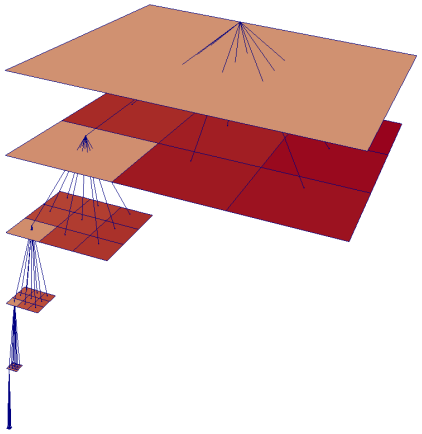
Prior to a discussion of Peano's data model, it might make sense to load all the files `grid-level-?-0.vtk` into your favourite visualisation software. Dilate them according to their level encoded in the name. In Paraview, you have to select each file, apply Filters/Alphabetical/Transform, and dilate each file by -0.2 along the z-axis per level. You might end up with something alike:



Again, feel free to create a video that shows how additional levels are added in each step. There's a more elegant way to end up with a similar picture. I once created a graph visualiser that is today also available as predefined mapping. Just modify your adapters as follows:

```
adapter:
  name: CreateGridAndPlot
  merge-with-user-defined-mapping: CreateGrid
  merge-with-predefined-mapping: VTK2dTreeVisualiser(tree,getLevel)
```

This time, creating a video should be straightforward.



We see that speaking of Peano in terms of a spacetree software is only one way to go. We also could speak of it of a software managing Cartesian grids embedded into each other. The latter point of view reveals an important fact: Peano handles vertices and cells. Cells are embedded into each other as they form the tree and there is a clear parent-child relation. Vertices connect cells. A vertex is unique due to its position in space plus its level, i.e. there might be some coordinates that host multiple vertices. In math, we would call this generating system. There are two types of vertices: the standard ones are adjacent to 2^d cells on the same level with d being the dimension of the problem. All other vertices are hanging nodes.

> **Remark:** Hanging nodes are not stored persistently. They are created and destroyed in each traversal upon request and never stored in-between two iterations. You can never be sure how often a hanging nodes is constructed per traversal. It could be up to $2^d - 1$ times. You only know it is created once at least.

Cells and vertices hold data. This data is described in the files `Vertex.def` and `Cell.def`. The .def files feed into our tool DaStGen that translated them internally into a standard C++ class (though it does some more stuff: it autogenerates all MPI data types that we need later on for parallel codes, and it in particular compresses the data such that a bool field is really mapped onto a single bit, e.g.).

So what we might do now is the following. Open the `Cell.def` and edit it accordingly. Rerun the PDT, recompile, and work.

```
Packed-Type: short int;


class myproject::dastgen::Cell {
  discard parallelise int myNonPersistentInteger;
  persistent parallelise double myValue;
  persistent parallelise bool myBool1;
  persistent parallelise bool myBool2;
};
```

3.4 Counting vertices

4 Logging, statistics, assertions

4.1 The user interface

4.2 Repository fields

4.3 Log filter

4.4 Using logging and tracing

4.5 Statistics

4.6 Assertions

5 Applications

5.1 Matrix-free Jacobi solver

5.2 Shallow water code

5.3 Molecular dynamics

6 High performance computing

6.1 Multicore support

6.2 MPI with spacetree-associated data

6.3 MPI with data on the heap

6.4 Multicore load balancing

6.5 MPI load balancing