Atlas Graphics

Tutorial

- Mission
- Control Structure Design
- Available AbstractScenes
- Simple Use
- Adding New Plottable
- PlottableCollection
- Future Development
- References

J.Hrivnac, Atlas Software Meeting, 30Oct97

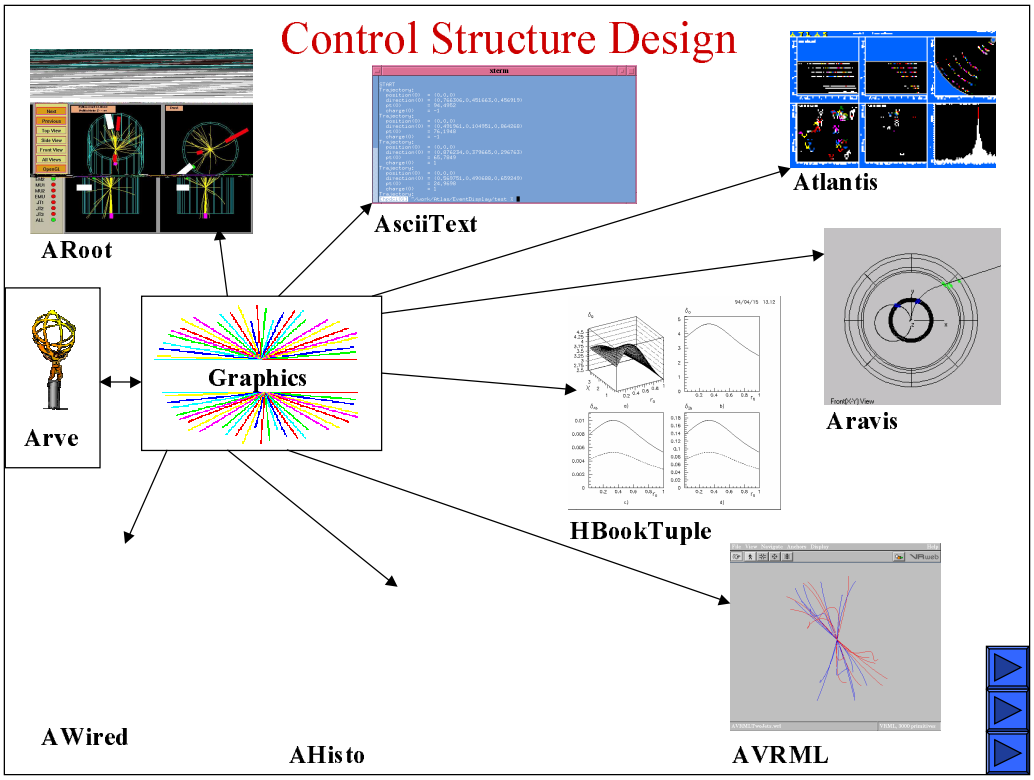
http://www-hep.fzu.cz/~hrivnac/Graphics/AtlasTutorial.1/

- expect more design diagrams, less c++ code, no flashing animations
- design is part of Atlas Graphics Design
- code runs and is available (see reference later)
- design diagrams and code fragments are sometimes simplified versions of the real stuff
- everything will evolve (especially code), but main ideas will stay
- you can play with it already now, it will be useful around end of the year
- users involvement may speed it up as interfaces to all other software domains are involved

Mission

The aim of the Atlas Event Display is to enable visual representation of the objects existing in the Atlas software. The Design of the Atlas Event Display is based on the believe that both requirements and graphics software abilities will be very broad at any time and will constantly evolve. The Atlas Event Display should be able to accommodate all that diversity and changes. This can be accomplished only by extreme **flexibility** and **modularity** of the core control structure. The Atlas Event Display is part of the full Atlas software package. The alpha version should be ready in 1997, beta version in 1998.

- first some general ideas about the Atlas Graphics
- our aim is not to choose THE right graphics package, but to interface various useful packages



- we deal with “Graphics” box and all the arrows
- some parts exists and are already interfaced, “proof of existence” has been made on others
- it will grow, some parts will die

- class diagram (classes, associations, multiplicity, methods, variables)
- core part of Graphics Control Structure
- for each Plottable, one PlottableModel and many PlottableRep exist
- PlottableModel mediates access to Plottable features useful for displaying
- only model() in Plottable so it is not polluted
- the real graphics implementation goes on between PlottableRep and AbstractScene
- AbstractScene is interface to any kind ov View (mostly graphical: 3D Event Display, Histogram,...)
- AbstractScene is a way how to look at the data (objects, Plottables)
- all available AbstractScenes can be registered in SceneList
- when Plottable is sent into SceneList, it is displayed on all AbstractScenes registered there
- Plottables and AbstractScenes can be removed or updated in SceneList
- most classes are abstract (they don’t implement, only declare), subclasses should implement

- Object Interaction Graph (two ways) (objects of classes, messages, sequence, returned objects)
- Plottable is displayed on AbstractScene by adding both into the same SceneList

- OIG (returned objects)
- messages passing between components of core control structure
- aModel asks its subclass (which has rep() method implemented) for PlottableRep, it knows that rep() is there, but doesn’t know (alone), how it is implemented
- aScene.display(PlottableRep* aRep) does the real graphics work, different for every AbstractScene

✓

Simple Use

```
// Create Plottable objects
Trajectory* electron = new Trajectory;
Trajectory* positron = new Trajectory;
// Create AbstractScenes
AsciiText anAsciiText("Scene1.txt");
VRMLFile aVRMLFile("Scene2.wrl");
HBTuple aTuple("Scene3.hbook", 11);
// Create SceneList
SceneList aSceneList;
// Register AbstractScenes in SceneList
aSceneList.add(&anAsciiText);
aSceneList.add(&aVRMLFile);
aSceneList.add(&aTuple);
// Display Plottables on registered AbstractScenes
aSceneList.add(electron);
aSceneList.add(positron);
```

- to use it is very simple (more sophisticated ways of using will be developed later)
- see Trajectory later (it is something like real track)
- AsciiScene writes output file which textual representation of Plottables
- VRMLFiles writes output file with VRML 3D representation of Plottables, they can be viewed by any VRML browser (e.g. Netscape)
- HBTuple creates standard Paw ntuple (mostly one row from each Plottable)
- things are only registered in SceneList (pointers or references go in)

✔

Adding New Plottable (Plottable)

```
class Plottable {
public:
  virtual PlottableModel* model() = 0;
};
```

```
class Trajectory : public Plottable {
public:
  Trajectory(Trajectory& tParticle;
    MagneticField& mField);
  TrackedParticle particle(float l);
  virtual PlottableModel* model();
private:
  Trajectory* _tParticle;
  MagneticField* _mField;
  TrajectoryModel* _tModel;
};
```

```
Trajectory::Trajectory(
  Trajectory& tParticle,
  MagneticField& mField):
  _tModel(0),
  _tParticle(&tParticle),
  _mField(&mField) {
}
```

```
PlottableModel* Trajectory::model() {
  if(! _tModel) {
    _tModel = new TrajectoryModel(*this);
  }
  return(_tModel);
}
```

```
TrackedParticle Trajectory::particle(float l) {
  Trajectory p =
    _mField->goToLength(*_tParticle, l);
  return(p);
}
```

- left column: header files (*.h)
- right column: source (*.cxx)
- yellow box: superclass
- blue box: class implemented by user (developer of Trajectory)
- e.g. abstract model() is implemented in subclass
- only parts of code shown here

- LocalHelix: only geometrical properties of particle
- TrackedParticle: real particle, it owns LocalHelix
- Trajectory: the whole track, can give TrackedParticle at any point in space, owns TrackedParticle
- MagneticField: moves TrackedParticle through the space
- trajectory is object of Trajectory
- don't look at PlottableCollection yet
- classes developed for testing of graphics, Trajectory may be usefull, MagneticField is wrong

- back in code:
- Trajectory is generated from TrackedParticle, it also needs MagneticField
- _tParticle and _mField is initialised during creation of TrackedParticle, _tModel only on request
- Trajectory implements model() defined in its superclass Plottable
- particle(float l) gives TrackedParticle at a length l from the origin, it uses MagneticField

Adding New Plottable (PlottableModel)

```
class PlottableModel {
public:
  virtual PlottableRep* rep(AbstractScene& aScene)
      = 0;
  virtual Plottable* plottable() = 0;
  virtual void plot(AbstractScene& aScene);
};
```

```
class TrajectoryModel : public PlottableModel {
public:
  TrajectoryModel(Trajectory& trajectory);
  virtual PlottableRep* rep(AbstractScene& aScene);
  virtual Plottable* plottable();
  Hep3Vector& position(float length);
  // Also direction, charge,...
private:
  Trajectory* _trajectory;
  PlottableRep* _pRep;
};
```

```
TrajectoryModel::TrajectoryModel (
  Trajectory& trajectory):
  _trajectory(&trajectory) {
}
PlottableRep* TrajectoryModel::rep(
  AbstractScene& aScene) {
  switch(aScene.type()) {
    case HBOOKTUPLE:
      _pRep = new HBTupleTrajectory(*this);
      break;
    // Other possibilities
  };
return(_pRep);
}
```

```
Plottable* TrajectoryModel::plottable() {
  return(_trajectory);
}
```

```
Hep3Vector& TrajectoryModel::position(float length) {
 return(_trajectory->particle(length).position());
}
```

- look at the core control structure, PlottableModel and PlottableReps should be created
- lets implement AsciiText, VRML and HBTuple

- first, lets look at the PlottableModel
- PlottableModel should have links to Plottable and PlottableRep corresponing to requested AbstractScene (kind of view)
- I'm showing one possible way how to do it, there may be better ways
- link to Plottable (Trajectory) is simple - it is obtained during creation
- link to PlottableRep is obtained when someone asks for it (he should supply AbstractScene), now ugly way of deciding is implemented - switch using AbstractScene.type(), in future better way will be implemented - MultiMethods, there will be no need to mention any particular PlottableRep or AbstractScene in PlottableModel (and so independent sharable libraries can be created and loaded when necessary)
- TrajectoryModel can give useful things from Trajectory (position, direction, charge,...), it is simple here, but sometimes it can do a bit of computing

Adding New Plottable (AsciiPlottableRep)

```
class AsciiPlottableRep : public PlottableRep {
public:
  virtual void text(ostream& out) = 0;
};
```

```
class AsciiTrajectory : public AsciiPlottableRep {
public:
  AsciiTrajectory(TrajectoryModel& aModel);
  virtual void text(ostream& out);
private:
  TrajectoryModel* _pModel;
};
```

```
AsciiTrajectory::AsciiTrajectory(
  TrajectoryModel& aModel) :
  _pModel(&aModel) {
}
```

```
void AsciiTrajectory::text(ostream& out) {
  out << "Trajectory:" << endl;
  out << "  position(0) = " << _pModel->position(0) <<
      endl;
  // Other quantities follow
}
```

- now, let's create AsciiTrajectory (kind of AsciiPlottableRep) which is used to write out text from the objects; AsciiText AbstractScene mediates this

- when AsciiText AbstractScene is asked to display PlottableRep, it simply asks it to do the output itself it calls AsciiPlottableRep::text(ostream& out)
- so the only requirement for AsciiPlottableRep is to implement text(ostream& out)

- so AsciiTrajectory implements text(ostream& out) declared in AsciiPlottableRep, user can deside, which output she wants here
- it uses _pModel for accessing Plottable features; it has got PlottableModel during creation
- this is one possibility how to design PlottableRep - AbstracScene interplay, you will see other in VRML part

Adding New Plottable (HBTuplePlottableRep)

```
class HBTuplePlottableRep : public PlottableRep {
public:
  virtual void fill(HBookFile* tFile) = 0;
};
```

```
class HBTupleTrajectory :
  public HBTuplePlottableRep {
public:
  HBTupleTrajectory(TrajectoryModel& aModel);
  virtual void fill(HBookFile* tFile);
private:
  TrajectoryModel* _pModel;
};
```

```
HBTupleTrajectory::HBTupleTrajectory(
  TrajectoryModel& aModel) :
  _pModel(&aModel) {
}
```

```
void HBTupleTrajectory::fill(HBookFile* tFile) {
  HepTuple* tuple = tFile->ntuple("trajectory", 1);
  tuple->column("x", _pModel->position().x(), 0);
  tuple->column("y", _pModel->position().y(), 0);
  tuple->column("z", _pModel->position().z(), 0);
  // Add other columns
  tuple->dumpData();
}
```

- HBTuple is very similar to AsciiText, it just writes different kind of file (Zebra Hbook file)
- so just replace text(ostream& out) with fill(HBookFile* tFile), tFile is defined in HBookTuple AbstractScene
- one should declare, fill and dump tuple in fill(...) function - one can define here the actual content on her ntuple
- very similar interface exists for both Root and HistOOgram (lhc++), restrictions are with compiler (HBookTuple runs only on g++ due to cernlib, Root runs on aCC due to shared library limitation on HP, HistOOgram runs only on CC due to Objectivity)

Adding New Plottable (VRMLRep)

```
class VRMLRep :
  public PlottableRep,
  public ThreeDSeparator {
public:
};
```

```
class VRMLTrajectory : public VRMLRep {
public:
  VRMLTrajectory(TrajectoryModel& aModel);
private:
  TrajectoryModel* _pModel;
};
```

```
VRMLTrajectory::VRMLTrajectory(
  TrajectoryModel& aModel) :
  _pModel(&aModel) {

  SFColor rgbRed(1, 0, 0);
  SFColor rgbBlue(0, 0, 1);
  SFColor rgb = ( _pModel->charge(0) > 0 ? rgbRed :
    rgbBlue );
  Arrow* arrow;
  for(float i = 0, l = 0; i < 50; i++, l += 0.01) {
    add(arrow = new Arrow(_pModel->position(l),
      _pModel->direction(l),
      0.01,
      _pModel->pt(l) / 100000,
      rgb));
    }
}
```

- VRMLFile AbstractScene creates file containing description of Plottables in 3D space (using VRML language)
- VRMLRep design is different to AsciiText or HBTuple: it doesn't do things via function implemented in the PlottableRep
- there are classes known to VRMLFile AbstractScene - ThreeDNodes, VRMLFile knows how to create VRML textual file for every ThreeDNode
- so it's enough for VRMLRep to just take form of ThreeDNode (it means to inherit from it) to be directly usable by VRMLFile
- the choice is to inherit VRMLRep from ThreeDContainer, because it can contain other ThreeDNodes which allows easy creation of complex VRML objects (ThreeDSeparator is just special kind of ThreeDContainer)

- so, it someone asks VRMLFile AbstractScene to display PlottableRep, it asks VRMLRep to do that (this is general rule) and it asks its superclass ThreeDContainer, which knows (from definition) how to do that

- VRMLRep can just contain any existing ThreeDNode (as it is container), it has add(ThreeDNode* function from ThreeDContainer to add any ThreeDNode to itself)
- VRMLFile automaticaly knows, how to handle such collections of ThreeDNodes
- VRMLTrajectory containes Arrow (which is complex ThreeDNode)
- on creation, VRMLTrajectory creates Arrow using obtained Trajectory values (it uses charge to define color of the Arrow,...)
- VRMLWindow is implemented in similar way, it uses external VRML browser (VRWeb) to display ThreeDNodes online; also Root or IrisExplorer can be in principle used for that

Adding New AbstractScene (AsciiText)

```
class AsciiText : public AbstractScene {
public:
  AsciiText(const HepString& fileName);
  virtual SceneType type();
  virtual void display(PlottableRep* aRep);
private:
  ofstream _out;
};
```

```
AsciiText::AsciiText(const HepString& fileName) :
  _out(fileName) {
  _out << endl;
  _out << "START" << endl;
}
```

```
SceneType AsciiText::type() {
  return(ASCIITEXT);
}
```

```
class AsciiPlottableRep : public PlottableRep {
public:
  virtual void text(ostream& out) = 0;
};
```

```
void AsciiText::display(PlottableRep* aRep) {
  ((AsciiPlottableRep*)aRep)->text(_out);
}
```

- let's look, how new kind of AbstractScene can be added, the simplest one is AsciiText
- you have already seen the graphs
- the way how to do that should be defined between AsciiText AbstractScene and AsciiPlottableRep
- AsciiText AbstractScene should only have standard methods (display, remove,... and type() which gives just ASCIITEXT enum type - will not be needed in future)

- from AsciiText::display(PlottableRep* aRep), aRep.text(ostream& out) is called

- so AsciiText should also define and store ostream _out
- AsciiText also does some simple initialization of the file

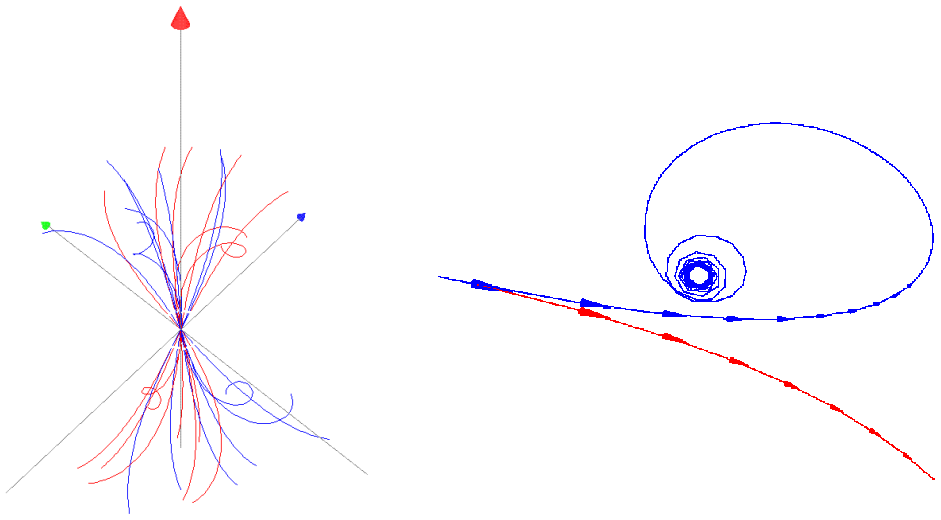
PlottableCollection

```
// Create several Trajectories
Trajectory* particle = new Trajectory[n];
// Initialise Trajectories from existing TrackedParticles
// ...
// Create jet from Trajectories
PlottableCollection simpleJet;
  for(int i = 0; i < n; i ++) {
    simpleJet.add(&t[i]);
  }
// Display simpleJet
aSceneList.add(&simpleJet);
```

- to enable reuse of existing Plottables, several additional concepts have been developed
- one of them is PlottableCollection
- PlottableCollection is Plottable (so it can be used in Graphics) and is collection of Plottables itself (so it can containe any other, already iplemented Plottable)
- as everything is known about Plottables contained in PlottableCollection (their PlottableModels and PlottableReps), there's no need to implement additional PlottableModel and PlottableReps for each new PlottableCollection
- there is just one PlottableCollectionModel which loops over all Plottables contained in PlottableCollection and gets their PlottableModels and PlottablesReps which are then feed into corresponding AbstractScenes

- so, for example, if you have several Trajectories, you can create simpleJet out of them by simple adding all those Trajectories into PlottableCollection
- there is no need for any other class to be defined, such simpleJet can be straight added into SceneList and displayed

PlottableCollection (example)



- VRMLFile AbstractScene creates file containing description of Plottables in 3D space (using VRML language)
- VRMLRep design is different to AsciiText or HBTuple: it doesn't do things via function implemented in the PlottableRep
- there are classes known to VRMLFile AbstractScene - ThreeDNodes, VRMLFile knows how to create VRML textual file for every ThreeDNode
- so it's enough for VRMLRep to just take form of ThreeDNode (it means to inherit from it) to be directly usable by VRMLFile
- the choice is to inherit VRMLRep from ThreeDContainer, because it can contain other ThreeDNodes which allows easy creation of complex VRML objects (ThreeDSeparator is just special kind of ThreeDContainer)

- so, it someone asks VRMLFile AbstractScene to display PlottableRep, it asks VRMLRep to do that (this is general rule) and it asks its superclass ThreeDContainer, which knows (from definition) how to do that

- VRMLRep can just contain any existing ThreeDNode (as it is container), it has add(ThreeDNode* function from ThreeDContainer to add any ThreeDNode to itself)
- VRMLFile automaticaly knows, how to handle such collections of ThreeDNodes
- VRMLTrajectory containes Arrow (which is complex ThreeDNode)
- on creation, VRMLTrajectory creates Arrow using obtained Trajectory values (it uses charge to define color of the Arrow,...)
- VRMLWindow is implemented in similar way, it uses external VRML browser (VRWeb) to display ThreeDNodes online; also Root or IrisExplorer can be in principle used for that

Future Development

Besides obvious improving, compliting, extending and debuging:

- Replacing switch in PlottableModel with MultiMethods.
- Adding ObjectBrowser on top of SceneList.
- Enabling access to AbstractScenes as Modules if Arve Control based on Modules.

- getting rid of switch in PlottableModels will allow creating independent shared libraries, which will allow loading of only really used subpackages; quite sophisticated MultiMethods (developed by Lassi Tuura) will be used
- ObjectBrowser will add possibility of using high level visual access to available objects (Plottables), for every Plottable one will immediately see available AbstractScene; it is like modern file managers (see TkFile or MS File Explorer) - only instead of files and directories one has objects and containers; Objectivity can be used to implement that
- depending on which structure will be chosen for Atlas Arve Control Structure, Graphics will offer any envelope; if modular structure is selected (like in Iris Explorer or SUN Java Studio), AbstractScenes will become Modules

References (SRT)

CVSROOT=/afs/cern.ch/atlas/software/cvs
$CVSROOT/offline/graphics

Already available

- ED - **Control Structure**
  - ED, src
  - EDU, srcU
- AsciiText - **Textual output**
  - AsciiText, src
  - AsciiTextU, srcU
- AVRML - **VRML 3D graphics**
  - AVRML, src
  - AVRMLU, srcU
- HBookTuple - **Paw ntuples**
  - HBookTuple, src
  - HBookTupleU, srcU
- test
  - test, src

Soon available

- Atlantis - **Special graphics**
  - Atlantis, src
  - AtlantisE, srcE
  - AtlantisU, srcU
- Aravis - **Arve interactive graphics**
  - Aravis, src
  - AravisE, srcE
  - AravisU, srcU
- ARoot - **Root**
  - ARoot, src
  - ARootU, srcU
- AHisto - **lhc++ HistOOgram**
  - AHisto, src
  - AHistoU, srcU
- AWired - **WWW Event Display**
  - AWired, src
  - AWiredU, srcU
- others

- consolidated code is available from standard Atlas SRT CVS code repository
- in principle, each subdomain (subdirectory) has six subdirectories: two for AbstractScene, PlottableRep etc., two for particular PlottableReps and two for external package (one is for headers, one for source)
- so up to three libraries are created from every subdomain: interface, users code and external code, shared libraries will be available
- dependencies are not implemented well till now - you should use everything, you cannot choose
- only g++ works, aCC will work soon

References (WWW)

- This Tutorial:
  http://www-hep.fzu.cz/~hrivnac/Graphics/AtlasTutorial.1/
- Atlas Graphics Group:
  http:// atlasinfo.cern.ch/Atlas/GROUPS/GRAPHICS/
- Atlas Graphics Design:
  http://atlasinfo.cern.ch/Atlas/GROUPS/GRAPHICS/Texts/EventDisplay/

- this tutorial is available on the Web
- there is also page of the Atlas Graphics Group
- it contains full design document, implementation guidelines (manual), hyperised code and picture gallery