D 5.4

API AND FINAL VERSION OF CAMERA SYSTEM

Project Number	FP7-ICT-231824
Project Title	Integrating Research in Interactive Storytelling (NoE)
Deliverable Number	D5.4
Title of Deliverable	API and Final Version of Camera System
Workpackage No. and Title	WP5 - Cinematography
Workpackage Leader	UNEW
Deliverable Nature	Report
Dissemination Level	Public
Status	Final
Contractual Delivery Date	31th December 2010
Actual Delivery Date	15th December 2010
Author(s) / Contributor(s)	Guy Schofield (UNEW), Patrick Olivier (UNEW) Marc Christie (INRIA)
Number of Pages	62

DOCUMENT HISTORY

Version	Date	Comment	Author(s)
1.1	December 15, 2010	Draft version for approval	UNEW: Guy Schofield and Patrick Oliver; INRIA: Marc Christie, Christophe Lino

TABLE OF CONTENTS

1. Description and content of Deliverable D5.4

This deliverable is a ***technical*** contribution to the IRIS Network of Excellence. It describes into details the APIs (Application Programming Interfaces) of our real-time cinematography system depicted in Deliverables D5.2 and D5.3.

The APIs is composed of a collection of libraries all written in C++ language (for Microsoft Visual Studio 2008). The DVD delivered with this document contains all the sources of our APIs, as well as the detailed documentation (over 400 pages of commented sources). The project is composed of over 80 C++ classes and approximately 15000 lines of code and is the result of our two years of research in the IRIS project.

1.1. Content

This deliverable will briefly describe how our Virtual Cinematography system can be employed in the context of an Interactive Storytelling application (Section 2.0), then provide a detailed description of the configuration files related to the API (Section 3.0), and finally a section describing the main classes of the API (Section 4.0).

1.2. Scientific and technical contribution

As detailed in deliverables D5.2 and D5.3, this software provides a complete and expressive approach to virtual camera control. Our contribution compares favourably to most approaches which generally emphasize on specific aspects of camera control: composition, offline editing, idiom-based approaches, annotated motions, path-planning and discourse. None have gathered the simultaneous abilities to perform real-time viewpoint selection, editing, composition and path-planning between regions of the environment with facilities to interactively control directorial style with a set of high-level cinematographic parameters (pacing, dynamicity, composition, preferred viewpoints) and means to control narrative dimensions (dominance, affinity, isolation). Furthermore, the expressiveness of the approach offers facilities to implement new narrative dimensions (such as shot intensification).

2. Interactive Cinematography v1.0 – A user Guide

2.1. General architecture and workflow

The Interactive Cinematography component is structured around a central class, the **Editor**. The **Editor** takes as input a **Story** (which is a set of **Events**), a **Topological Graph (**which contains all the geometry of the environment), a set of key targets called **Actors,** and a **Style**, and produces a sequence of camera viewpoints through calls to the **Editor::update(double dt)** method which returns the newly computed camera configuration, given the style and the constraints of the environments (events and visibility of key targets).

At each step, a call to **Editor::update(double dt)** performs the following steps:

1. Checks whether the camera should be maintained or whether a cut should be performed due to a loss of visibility of a key target, or due to the pacing constraints. The **method Editor::canMaintainCamera()** is in charge of this process.

2. If the camera is to be maintained, a reconfiguration process is called which re-computes an appropriate camera location/orientation using a **ConfiguringProcess** object with method **ConfiguringProcess::reconfigure(Camera* cam, double dt).** Here the purpose is compute for the next **dt** a new configuration inside the same Director Volume or along a camera path.

3. If the camera is not to be maintained (i.e. a cut or a transition is necessary) the **Editor** looks at all the **Events** in the **Story** occurring at the current moment and selects the most relevant ones with **Story::currentEventsSortedByRelevance(Date &d)**. From there:

    a. The **DirectorVolumes** are computed for the most relevant **Event** (the **DirectorVolumes** is the set of all the visible viewpoints around the **Event**). Given the nature of the **Event,** one, two or more characters are involved and the spatial partitions are built into **DirectorVolumes** for one or two characters or more characters. The **DirectorVolumes** are computed using a **FilteringProcess** which takes as input the current event**: FilteringProcess::setEvent(Event *)** and generates the volumes given the key-targets and their visibility in the 3D environment.

    b. Different **Filters** are then applied, depending on whether a cut needs to be performed to switch to another **DirectorVolume**, or whether a transition needs to be performed. For cuts, the filtering process performed by **FilteringProcess::filter(FilteringProcess::CUT)** and for transitions by **FilteringProcess::filter(FilteringProcess::TRANSITION).**

    c. Both **Filters** return a **DVList**. A **DVList** is a list of Director Volumes (possible viewpoints where the camera can cut to, or move to in a transition). The filtering process applies a sequence of **Filters** (a subset of **AreaFilter, CoherencyFilter, CurrentSVFilter, DistanceFilter, FramingFilter, GoalFilter, IdiomFilter, IFilter, IntensityFilter, LoAFilter, LoIFilter, NearestFilter, RandomIdiomFilter, SubdistanceFilter, TargetFilter, ThirtyDegreeFilter and VisibilityFilter**). The exact sequence is defined and implemented in the FilteringProcess class (see **FilteringProcess.cpp**).

d. The FilteringProcess returns a **DVSortedList** (which is a sorted list according to costs defined for each a couple of DirectorVolumes source and destination: **Float Filter::cost(DVTuple*))**. From such a **DVSortedList,** the first **DirectorVolume** is extracted, and a **ConfiguringProcess** is applied which selects the best camera configuration within the DirectorVolume, through the method **ConfiguringProcess::configureCut()** or **ConfiguringProcess::configureTransition(),** given the specific situation.

4. In specific cases where no solutions are found with the current filters (e.g. the problem is too constrained), we resort to a default strategy which consists in searching for an appropriate extreme long shot **DirectorVolume**, by calling **FilteringProcess::filter(FilteringProcess::EXTREME_LONG)**. Cutting to extreme long shots is generally less noticeable than directly violating main continuity rules. And spatial coherency in the specific case of extreme long shots does not need to be maintained.

2.2. Commented examples

2.2.1. Basic use

The first example illustrates a simple use of the API. The process requires only five lines of code to create a demo. In detail, the Interactive Cinematography v1.0 API requires:

- loading the topological representation of the 3D environment (in Topoplan env3d format)

```
Topology::Graph::load ( “topology”, “topology_character” ) ;
```

  - “topology” is a filename referring to the internal representation of the 3D scene suitable for camera motion planning (in .env3d format)
  - “topology_character” is a filename referring to the internal representation of the 3D scene suitable for character motion planning (in .env3d format)
  - two distinct environments are necessary since the camera may navigate above obstacles (e.g. tables) while the characters may not, for the same geometry
  - The env3d format is directly generated from Ogre XML representations using Topoplan tool.

- creating a Demo object and an Editor

```
Demo *demo = new MyDemo();

Editing::Editor *editor = new Editing::Editor(*demo, editedOgreCamera);
```

```
Editing::Style &style = editor->style();
```

- The Demo interface only contains some primitive methods to display topological representations, Director Volumes, camera paths (mainly for debugging purposes). A new application needs to derive and implement its own Demo instance (here MyDemo) or reuse an existing demo (we provide a DemoBOgre instance which uses Ogre animation engine)

- The Editor is then constructed with an instance of demo. An instance of the Style class is directly created by the editor, and accessed via a getter method (Editor::getStyle()). The variable `editedOgreCamera` represent the Ogre::Camera that is synced with the Editing::Camera used in the Editor.

- All modifications are then possible on the Style (shot duration, visibility, range of camera dynamicities, degree of intensity, narrative dimension, dutch view and atmosphere)

- Creating characters involved in the story. Facilities are provided to handle a character defined by an Ogre::Entity, but any representation can be employed by simply deriving from a GenericActor class and implementing the appropriate methods. Here we provide a context with two actors (named Symes and Smith)

```
Semantic::Actor *Symes = new AnimActor("Symes",
symesOgreBodyBoundingBox, symesOgreHeadBoundingbox, symesConvexhull);

Symes->setEyesBB(symesOgreLeftEyeBoundingBox,
symesOgreRightEyeBoundingBox);

Semantic::Actor *Smith = new AnimActor("Smith",
smithOgreBodyBoundingBox, smithOgreHeadBoundingbox, smithConvexhull);

Smith->setEyesBB(smithOgreLeftEyeBoundingBox,
smithOgreRightEyeBoundingBox);
```

- Variables `symesOgreBodyBoundingBox`, `symesOgreHeadBoundingbox`, `symesOgre[Left|Right]EyeBoundingBox` and `symesConvexhull` respectively represent the Ogre::Entity related to the body of the character, the Ogre::Entity related to the head of the character, the Ogre::Entity related to the left or right eye of the character, and the convex hull of the character defined as a vector of Vector2d points (see Demo/GenericActor.h)

- Adding events to the Story. The Event class provides a way to add new events to the story. Here for the purpose of the illustration, we create and insert a first event which lasts 2 seconds ("Symes looks at Smith") and a second which lasts 10 seconds ("Smith talks to

Symes"). An event is defined by its nature (what is the action), its start and stop time, the characters involved, and the relevance of the event (in case of parallel events).

```
Story::Event *event1 = new Story::Event("A looks at B", 0, 2,
ActorConfiguration(Symes, Smith), 5);

Story::add(event1);


Story::Event *event2 = new Story::Event("A talks to B", 2, 12,
ActorConfiguration(Smith, Symes), 5);

Story::add(event2);
```

- Variables Symes and Smith refer to the instances of AnimActor described above.

- And using the editor to update the camera at each frame:

```
// … somewhere in the update of the display

Editing::Camera &mycamera = editor->update(dt);

Ogre::Camera *myOgreCamera = editor->ogreCamera();
```

- The update(dt) method computes a new camera position given the Style, the environment and the events occurring and returns an instance of a camera, which is synced with the camera of the 3D engine.
- The last line of code is provided only for the purpose of the demonstration, but the update the Ogre::Camera has already be done in update(dt) method.

2.2.2. Elaborate use

Most of the methods in the Editor can be tailored for specific use by simply subclassing the Editor class and subclassing the main components: the FilteringProcess and the ConfiguringProcess.

```
class Editor {
      public:

            Editor(BOgreDemo& demo);
```

```
            ~Editor();

            void resetCamera();

            Camera& update(Float dt);

            void processCamera(Float dt);

            bool canMaintainCamera();
};
```

- the resetCamera() method enables to reset all components of the Editor
- the processCamera(double dt) is called by the update() method to compute the next location
- the canMaintainCamera() decides whether the current view should be maintained or not

All the same, the class FilteringProcess can be modified (e.g. by implementing specific ways of performing the filtering process)

```
class FilteringProcess {
      public:

           FilteringProcess(Editing::Editor& ed);

            ~FilteringProcess();

           Filters::DVList* filter(Directive d=CUT);

};
```

- The filter(Directive) method performs the filtering process (application of all the filters. Directive specifies the type of the filter to be performed (for a CUT or for a TRANSITION between DirectorVolumes). The method can be reimplemented to change the default behaviour of the filtering process.

The class ConfiguringProcess can be altered to handle the computation of a camera configuration in a different way inside a Director Volume.

```
class ConfiguringProcess
{
      public:

            ConfiguringProcess(Editor&);

            ~ConfiguringProcess();
```

```
            void configureCameraInVolume(Camera &,
                 const Semantic::VolumeParameters &,
                 Base::ConvexCell *, std::string &,
                 Semantic::ActorConfiguration &);


            void reconfigure(Camera&, Float dt=0);

            void configureCut(Camera &c, Filters::DVTuple &t);

            void configureTransition();
};
```

- The configureCameraInVolume() enables the recomputation of a camera configuration inside the current DirectorVolume (identified here as a ConvexCell)
- The reconfigure() enables the recomputation of a camera configuration
- The configureCut() enables the recomputation when a cut is performed (selects a configuration in the target Director Volume and computes the appropriate composition.

3. Description of configuration files

To ease the development and configuration of the application, most of the parameters are directly stored in XML representation files. Precisely, we have defined:

- The “Events” file: a file which describes the sequence of events occurring (in a non interactive context). The file is mainly for simulating actions occurring. In an interactive context, the Events are generated on the fly.
- The “Viewpoints” file: a file which stores the mapping between an event type (e.g. “character A looks at character B”) and the specific viewpoints and camera motions allowed to portray the action (section 3.2)
- The “Composition” file: a file which stores the values of the composition process for the Director Volumes (section 3.3). This avoids the burden of redefining the framing parameters each time a composition process is performed.
- The “Configuration” file: a file which refers to all the other files and contains elements referring to the characters involved in the scene, the topology and geometry, the sound tracks and the directorial style parameters.

All three files are detailed in the following paragraphs.

3.1. The “Events” file

```
<?xml version="1.0" ?>
<Events>

      <Action name="A pours gin" begin="0" end="4" relevance="1">
            <Character name="Smith" />
      </Action>

      <Action name="A drinks gin" begin="4" end="12" relevance="1">
```

```
            <Character name="Smith" />
      </Action>

      <Action name="A speaks to B" begin="18" end="21" relevance="4">
            <Character name="Smith" />
            <Character name="Syme" />
      </Action>

      <Action name="A comes" begin="83" end="96" relevance="3">
            <Character name="Parsons" />
      </Action>

<Events>
```

The events file is provided for convenience (i.e. simulating actions occurring in the environment). In an interactive storytelling context, the events are generated on the fly as illustrated in section 2.2.1

3.2. The "Viewpoints" file

```
<?xml version="1.0" ?>
<Rules>

      <!-- 1 people -->

      <Action name="A comes" >
            <Behavior>
                  <Traveling/>
                  <Panning/>
            </Behavior>
            <Style dimension="default" >
                  <Idiom name="A-back-long" />
                  <Idiom name="A-front-long" />
            </Style>
            <Style dimension="long take" >
                  <Idiom name="A-34front-long" />
            </Style>
      </Action>

      <Action name="A drinks gin" >
            <Style dimension="default" >
                  <Idiom name="A-front-closeup" />
                  <Idiom name="A-front-close" />
                  <Idiom name="A-34front-closeup" />
                  <Idiom name="A-34front-close" />
                  <Idiom name="A-34back-closeup" />
                  <Idiom name="A-34back-close" />
            </Style>
            <Style dimension="long take" >
                  <Idiom name="A-front-close" />
                  <Idiom name="A-34front-close" />
                  <Idiom name="A-34back-close" />
```

```
            </Style>
      </Action>


      <!-- 2 people -->

      <Action name="A comes to B" >
            <Behavior>
                  <Panning/>
            </Behavior>
            <Style dimension="default" >
                  <Idiom name="Ext-A-close" />
                  <Idiom name="Ext-B-close" />
                  <Idiom name="Ext-A-closeup" />
                  <Idiom name="Ext-B-closeup" />
                  <Idiom name="Apex" />
                  <Idiom name="Par-A-close" />
                  <Idiom name="Par-B-close" />
                  <Idiom name="Par-A-mcu" />
                  <Idiom name="Par-B-mcu" />
            </Style>
            <Style dimension="isolation" >
                  <Idiom name="Int-A-closeup" />
                  <Idiom name="Int-A-close" />
                  <Idiom name="Int-A-mcu" />
                  <Idiom name="Par-A-closeup" />
                  <Idiom name="Par-A-close" />
                  <Idiom name="Par-A-mcu" />
                  <Idiom name="Par-A-long" />
            </Style>
            <Style dimension="long take" >
                  <Idiom name="Par-A-closeup" />
                  <Idiom name="Par-B-closeup" />
            </Style>
      </Action>
</Rules>
```

The viewpoints file specifies for each action (identified by its string **name**), the list of specific viewpoints which highlight the action. Most actions can be shot from any viewpoint, but certain specific ones may require a restricted set of viewpoints. For example, the Action “A comes” described in our example can be shot from front or back long shots. For each action, specific indicators may be specified:

- Behaviour field restricts the range of dynamic behaviours authorized for the camera (e.g. Travelling, Panoramic, etc)
- Style field restricts the range of shots (viewpoints) for a given style identified by its name (e.g. “long take”, “dominance”, “affinity”, “isolation”).

3.3. The “Composition” file

```
<?xml version="1.0" ?>
<Composition>

      <Viewpoint type="FRONT" shot="LONG" >
            <Style dimension="default" >
                  <Character framing_one="(+0.01;+0.25)" />
            </Style>
      </Viewpoint>

      <Viewpoint type="FRONT" shot="MEDIUM CLOSEUP" >
            <Style dimension="default" >
                  <Character framing_one="(+0.01;+0.25)" />
            </Style>
      </Viewpoint>

      <Viewpoint type="FRONT" shot="CLOSE" >
            <Style dimension="default" >
                  <Character framing_one="(+0.01;+0.25)" />
            </Style>
      </Viewpoint>

      <Viewpoint type="FRONT" shot="CLOSEUP" >
            <Style dimension="default" >
                  <Character framing_one="(+0.01;+0.25)" />
            </Style>
      </Viewpoint>

      <Viewpoint type="3/4 FRONT" shot="LONG" >
            <Style dimension="default" >
                  <Character framing_one="(+0.01;+0.25)" />
            </Style>
      </Viewpoint>

     ...

     <Viewpoint type="INTERNAL" shot="CLOSEUP" >
            <Style dimension="default" >
                  <Character framing_one="(+0.25;+0.25)"
                             framing_both="(+0.25;+0.25)" />
                  <Character framing_one="(-0.25;+0.25)"
                             framing_both="(-0.25;+0.25)" />
            </Style>
      </Viewpoint>

</Composition>
```

The composition file specifies how the characters should be framed for each shot, depending on the specified dimension. The framing settings are specified in terms of screen coordinates (where horizontal and vertical ranges are defined between -1 and 1). Since the nature of each shot determines the number of characters involved, compositions are specified for one or two characters. Each dimension can specify its on screen composition.

3.4. The “Configuration” file

A configuration file can be used to facilitate the loading of all the previous files and sets of parameters. We provide this file as an example. Related loader can be found in DemoLoaderBOgre.h/DemoLoaderBOgre.cpp. In an interactive context, the events file is not necessary (since events are generated by the system).

```
<?xml version="1.0" ?>
<Configuration>

      <!-- Environment Settings -->

      <Environment acceptable_dist="1e-3" alignement_factor="1e-2" >

            <!-- simplified TopoPlan subdivision used for the camera -->
            <Camera fileName="canteenScene" />

            <!-- TopoPlan subdivision used for the characters -->
            <Actor fileName="canteenScene" />

            <Scene fileName="LipSynchScene2" />
            <Scene fileName="Parsons5" />
            <Scene fileName="Julia" />
            <Scene fileName="BaldMale" />
            <Height minHeight="0" maxHeight="3" floor="0" />
      </Environment>


      <!-- Story settings -->

      <Idioms file="1984-idioms.xml" />

      <Events file="canteenScene/1984-events.xml" />

      <FrameComposition file="1984-composition.xml" />

      <Character
                        name="Smith" boundingbox="SmithHeadBox"
                        radius=".1" nbPoints="10"
                        recomputeVisibilityEvery="10" />
      <Character
                        name="Syme" boundingbox="SymeHeadBox"
                        radius=".1" nbPoints="10"
                        recomputeVisibilityEvery="10" />

      <!--  Camera settings  -->

      <ControlledCamera enable="true"
                                    display="true"
                                    display_SV="false"
                                    scale=".25"
                                    height="1.3" >
```

```
            <PathPlanning visibility_weight="1e2" waypoint_density="1"
nb_optimize_pass="2" />
            <Visibility min="0" max="1" />
            <Shot min="3" max="8" cutIntervalLength="2"/>
            <Dynamicity value="" />
            <Behavior>
            </Behavior>
            <Style dimension="long take" />
                  <Character name="Smith" />
            </Style>
      </ControlledCamera>


</Configuration>
```

4. API of Interactive Cinematography v1.0

4.1. Directory Hierarchy

Directories

This directory hierarchy is sorted in the following way:

include
- Base
- BSP
- Demo
- Director
- Editing
- Filters
- Planning
- Regions
- Semantic
- Story
- Topology
- Visibility

4.2. Namespace Index

Namespace List

Here is a list of all namespaces with brief descriptions:

4.3. Directory Documentation

Director Namespace Reference

Classes related to the concept of Director Volumes (ie Semantic Volumes + Visibility Volumes)

Classes

class Map
Describes a map merging both visibility and semantic information on a configuration of actors.
class Volume2d5
Describes a 2D5 volume merging both visibility and semantic information

---

Detailed Description

This namespace contains the classes related to the concept of Director Volumes. A Director Volume is a 2D5 convex cell merging both Semantic and Visibility information on the camera configurations included in that cell, for a given actor configuration. The Director Map contains the whole set of Director Volumes associated to such an actor configuration and is structured as follows. To each Topological Cell of the environment is associated a BSP Tree, whose leaves are the Director Volumes resulting from the Topological Cell partitioning.

Editing Namespace Reference

Editing of the movie in real-time (ie deciding when, where and how to cut/path-plan to a new viewpoint)

Classes

class Camera
Represents a 3D camera.
class ConfiguringProcess
Process that fix the parameters of a camera wrt a given 2D5 director volume and a set of constraints on the camera Mimics the behavior to the photography director and the camera operator. class CostFunction
class Editor
Reproduces the behavior of the editor.
class FilteringProcess
Process of filtering and annotating a set of Director volumes wrt a set of cinematic rules.
class FrameComposition
Structure for processing the farme composition of key elements on screen.
class Idiom
Represents the series of characteristic viewpoints related to conveying a given action.
class Style
Cinematic style
class Viewpoint

represent a characteristic viewpoint of the cinematography class XmlEditingManager

---

Detailed Description

This namespace contains the classes used for Editing the movie in real-time (ie deciding when, where and how to cut/path-plan to a new viewpoint). The Editor class is the entry point of that module. It will apply a cinematic style while planning the series of viewpoints and transitions. The Editor processes the next camera configuration as follows. First, the Editor will choose either to maintain the current camera behavior (static viewpoint, moving camera, etc) or to make a cut or continuous transition w.r.t. the editing style. Second, if making a cut or transition the editor chooses the next event to convey, then chooses the more appropriate viewpoint for that purpose by (1) filtering the Director Volumes, then (2) configuring the camera parameters within the resulting Director Volume. This configuration process uses a frame composition process to compose the key elements on screen and may also use the motion planning graph (i.e. Roadmap) for continuously moving the camera.

Filters Namespace Reference

Filtering/Annotation on Director Volumes.

Classes

class Annotation
- *Represent the set of annotations that are associated to Director Volumes while filtering them.*

class DVTuple
- *Represent a Director Volume with additional annotations.*

class IFilter
Represents a filter that work on a given criteria to annotate and prune some director volumes.

class AreaFilter
Filter that works on an area criteria.

class CoherencyFilter
Filter that works on a coherency criteria on the series of selected viewpoints.

class CurrentSVFilter
Filter that works on the diversification of viewpoints when making a cinematic transition.

class DistanceFilter
Filter that works on the change of distance when making a cut.

class IdiomFilter
Represents a filter that work on a preferred viewpoint selection.

class IntensityFilter
Filter that works on the intensification/relaxation in the cinematography.

class LoAFilter
Filter that works on the rule wrt the line of action of the cinematography.

class LoIFilter
Filter that works on the 180 degree (or Line of interest) rule of the cinematography.

class NearestFilter
Filter that works on a distance criteria.

class ThirtyDegreeFilter
Filter that works on the 30 degree rule of the cinematography.

class VisibilityFilter
- *Filter that works on a visibility criteria w.r.t. key subjects.*

Detailed Description

This namespace contains the implemented filters that work over Director Volumes. Each filter takes a set of annotated Director Volumes as input and output. They can prune and/or add annotations to the Director Volumes. Each one then associate a cost to the remaining Director Volumes, so that a filtering process is capable of combining them as an overall cost.

Planning Namespace Reference

Classes related to the path-planning for a camera.

Classes

class ArcInfo
Represents an arc of the camera motion planning graph, passing through a given director volume
class ArcValueFunction
Cost function of the camera motion planning graph.
class NodeInfo
Represents a node of the camera motion planning graph, described as a waypoint located on the frontier between two connected director volumes
class Roadmap
Represents the camera motion planning graph

Detailed Description

This namespace contains the classes related to the path-planning for a camera. The roadmap is built over the Director Volumes and can be accessed to plan a camera path between two camera configurations in the entire environment.

Semantic Namespace Reference

Classes related to the concept of Semantic Volumes (ie characteristic viewpoints)

Classes

class Actor
Represents the minimal information about an actor needed to create semantic and visibility volumes.
class ActorConfiguration
Represents a configuration of actors relative to a given action
class Tree
Represents the BSP tree that contains the semantic information relative to a given topological cell
class Volume
Represent a semantic volume, ie a characteristic viewpoint, represented as a 2D cell
class VolumeParameters
- *Represents the parameters of a semantic volume (Type of shot, distance from actors, side wrt the line of interest (LOI), side wrt actors)*

Detailed Description

This namespace contains the classes related to the concept of Semantic Volumes, ie characteristic viewpoints (for instance "Over the Shoulder medium shot", "APEX shot" on a couple of actors, or "Profile close-up shot" on a single actor) with an indication about the side of the camera wrt the Line of Interest. The Semantic Tree is a BSP Tree that is procedurally computed for a given actor configuration (one or two actors), with no consideration about the environment topology, and whose leaves are the Semantic Volumes.

Story Namespace Reference

Classes related to the unfolding story.

Classes

class Action
Represents a type of event which is likely to occur in the scene.
class Event
Represent an event that is occurring in the scene.

Functions

void add (Event *e)
Add an event to the story
std::vector< Event * > & allEvents ()
Get all the events of the story.
void clear ()
Clear all the events of the story
std::vector< Event * > & currentEventsSortedByRelevance (double date)
Get the currently unfolding events, sorted by their relevance.

Detailed Description

This namespace contains the classes related to the unfolding story.

Topology Namespace Reference

Classes related to the 2D5 navigable topology used by cameras.

Classes

class Cell
Represents a topological cell, used in the spatial partitioning of the environment
class Graph
Represents a topological Cell and Portal Graph (CPG)

class Portal
Represents a topological portal, used in the spatial partitioning of the environment.

Detailed Description

This namespace contains the classes related to the 2D5 navigable topology used by cameras. The environment topology is preprocessed as a Cell and Portal Graph *(*CPG) containing 2D5 convex Cells representing the free space for the camera (for instance rooms). The Cells are connected by Portals, representing the free 2D5 edges that are shared by adjacent Cells (for instance doors, windows, etc).

Visibility Namespace Reference

Classes related to the concept of Visibility Volumes (ie viewpoints from where key elements are visible)

Classes

class Map
Visibility map wrt an actor, computed on the whole environment.
class Volume2d5
Represents a visibility volume related to an actor. Typedefs

Enumerations

enum Categorization { TOTAL_VISIBILITY, TOTAL_OCCLUSION, PARTIAL_VISIBILITY, UNCATEGORIZED }

Detailed Description

This namespace contains the classes related to the concept of Visibility Volumes, ie viewpoints from where key elements (actors) are visible. A Visibility Volume is a 2D5 convex cell that carries Visibility information on the camera configurations included in that cell (total visibility, no visibility or partial visibility) on a given actor. The Visibility Map contains the whole set of Visibility Volumes associated to such an actor and is structured as follows. To each Topological Cell of the environment is associated a BSP Tree, whose leaves are the Visibility Volumes resulting from the Topological Cell partitioning.

Enumeration Type Documentation

enum Visibility::Categorization

Enumerator:

TOTAL_VISIBILITY
TOTAL_OCCLUSION
PARTIAL_VISIBILITY
UNCATEGORIZED

4.4. Class Documentation

Demo Class Reference

Define the basis for any demonstration, non dependent of a Rendering Engine.
`#include <Demo.h>`

Public Member Functions

void create3DSegment (Vector3d s1, Vector3d s2, Printer::SVG::Color &c)
void createPortal (const std::vector< Ogre::Vector3 > &tab, double height, double width, Ogre::ColourValue col)
void createVolume2d5 (const std::vector< Ogre::Vector3 > &tab, double height, double width, Ogre::ColourValue col)
Demo ()
void displayPath (std::vector< Vector2d > path, Printer::SVG::Color &c=Color::BLUE)
void displayPathNode (Vector2d pathNode, Printer::SVG::Color &c)
double getVolumesOpacity ()
~Demo ()

Public Attributes

double Demo_EndTime
double Demo_StartTime
bool Demo_Use_Path_Planning
bool Demo_Use_Semantic
bool Demo_Use_Visibility

Detailed Description

Define the basis for any demonstration, non dependent of a Rendering Engine. This is the top-level element and entry point of a demonstration. To create your own demonstration you must extends that class, and put in place all the needed elements used by the rendering engine you use.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/15

GenericActor Class Reference

Description of an actor for the demo.
`#include <GenericActor.h>`

Public Member Functions

virtual void displayOgre (Demo &out)
Displays the actor in Ogre.

GenericActor (std::vector< Vector2d > cloud)
Constructor.

virtual Vector3d getEyesPosition ()=0
Returns the position of the eyes of the actor.

virtual void reset ()=0
Resets the properties of the actor.

virtual void update (const double dt)=0
Updates the properties of the actor.

virtual ~GenericActor ()
Default destructor.

Protected Attributes

bool init
double m_height

Detailed Description

This represents the information any actor need to provide for any demo (position of the body, orientation of the body, position of its eyes, etc).

Story::Action Class Reference

Represents a type of event which is likely to occur in the scene.
`#include <Action.h>`

Public Member Functions

Action (std::string name)
Constructor.

void addIdioms (std::string style, std::vector< Editing::Idiom * > idioms)
Add a set of idioms to the action, wrt a given style.

void allow (std::string style_name)
Allow the camera to have a particular behavior (panning, traveling, intensity, etc) when conveying this kind of action. Warning: some of the behaviors are applied iff the camera style and the conveyed action allows it.

std::vector< Editing::Idiom * > & idioms (std::string style)
Returns the set of idoms associated to the action wrt a given style.

bool isAllowed (std::string style_name)
Whether the camera is allowed to have a particular behavior (panning, traveling, intensity, etc) when conveying this kind of action.

std::string name ()
returns the name of the action
~Action ()
Destructor.

Static Public Member Functions

static void clearAll ()
Clear all the known actions.
static Action * getAction (std::string name)
Search an action by its name.
static Story::Action * loadFromXML (TiXmlElement *pElem)
LOads an action form an XML file.

Static Public Attributes

static std::map< std::string, Action * > all_actions

Protected Attributes

std::set< std::string > m_allowed_styles
std::map< std::string, std::vector< Editing::Idiom * > > m_idioms
std::string m_name

Detailed Description

Represents a type of event which is likely to occur in the scene (for instance “A speaks to B”, “A walks to the table”, etc).

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Story::Event Class Reference

Represent an event that is occuring in the scene.
`#include <Event.h>`

Public Member Functions

Action * action ()
return the corresponding action
double begin ()
returns the begin time of the event

double duration ()
Returns the duration of the event.

double end ()
Returns the end time of the event.

Event (std::string action, double begin, double end, Semantic::ActorConfiguration config, int prio)
priority (the higher, the more relevant. the older event is favored when same priority).

Semantic::ActorConfiguration & involvedActors ()
returns the configuration of actors that is involved in the event

double percentage (double date)
returns the percentage of the event unfolding (0 <= p <= 1)

int priority ()
returns the priority of the event

std::string toString ()
Provides a string representation of the event.

~Event ()
Destructor.

Static Public Member Functions

static Event * loadFromXML (TiXmlElement *)
Loads an event from an XML file.

Protected Attributes

std::string m_action
end date of the event

double m_begin
Semantic::ActorConfiguration m_configuration
corresponding action type

double m_end
begin date of the event

int m_priority
configuration of actors

Detailed Description

Represent an event that occurs in the scene. It is an instance of Action (for instance “Syme speaks to Smith”, “Parsons walks to the table”), with a start and end date, and a relevance in the story context.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Editing::Editor Class Reference

Reproduces the behavior of the editor.
`#include <Editor.h>`

Public Member Functions

void _updateOgreCamera ()
const Camera & camera () const
get the current camera parameters
Camera & camera ()
get the current camera parameters
bool canMaintainCamera ()
check whether the camera location can be maintained
const Story::Event * conveyedEvent () const
get the currently conveyed event
Story::Event * conveyedEvent ()
get the currently conveyed event
Demo & demo ()
Get the associated demonstrator.
const Demo & demo () const
Get the associated demonstrator.
Editor (Demo &demo, Ogre::Camera *camera)
const Style & Editor::style () const
get the current style
Semantic::ActorConfiguration & frameActors ()
get the current configuration of framed actors
const Semantic::ActorConfiguration & frameActors () const
get the current configuration of framed actors
ConfiguringProcess * getConfiguringProcess () const
Returns the process used for configuring a camera in a volume.
FilteringProcess * getFilteringProcess () const
Returns the process used to select/annotate a set of director volumes.
double & lastTransitionDate ()
get the date of the last transition (cut or smooth transition)
const double & lastTransitionDate () const
get the date of the last transition (cut or smooth transition)
Ogre::Camera * ogreCamera ()
get the current Ogre camera parameters (the Ogre camera is linked to the edited camera)
const Ogre::Camera * ogreCamera () const
get the current Ogre camera parameters (the Ogre camera is linked to the edited camera)
void processCamera (double dt)
void resetCamera ()
Resets the camera edition to its initial state, just before beginning editing.
Style & style ()
get the current style
Semantic::Volume & SV ()

Get the currently used SV for conveying the chosen event.

const Semantic::Volume & SV () const

Get the currently used SV for conveying the chosen event.

Camera & update (double dt)

~Editor ()

Destructor.

Protected Attributes

Camera m_camera

the current camera parameters

ConfiguringProcess * m_configuring_process

the configuring process of the camera

Demo & m_demo

the associated demonstrator

Story::Event * m_event

the current conveyed event

FilteringProcess * m_filtering_process

the filtering process on director volume

Semantic::ActorConfiguration m_frame_actors

the current configuration of actors to frame

double m_last_transition_date

the date of the last cut

Ogre::Camera * m_ogre_camera

the corresponding Ogre camera

Style m_style

the current editing style

Semantic::Volume m_SV

memory of the currently used SV

Detailed Description

This is the class that reproduces the behavior of the editor. It decides when, where and how to cut or move the camera, according to the associated cinematic style and environment constraints.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Editing::Style Class Reference

Cinematic style.

```
#include <Style.h>
```

Public Types

enum Atmosphere { Stability, Instability }
enum Dynamicity { NO_DYNAMICITY = 0, ORIENTATION = DY_ORIENTATION,
POSITION_ORIENTATION = DY_ORIENTATION|DY_POSITION, FULL =
DY_POSITION|DY_ORIENTATION|DY_TRANSITION }
enum Intensity { NO_INTENSITY, INCREASE, DECREASE }

Public Member Functions

void allow (std::string style_name)

Allow the camera to have a particular behavior (panning, traveling, intensity, etc) when conveying the actions. Warning: some of the behaviors are applied iff the camera style and the conveyed action allows it.

std::set< std::string > allowedBehaviors ()
void disallow (std::string style_name)

Disallow the camera to have a particular behavior (panning, traveling, intensity, etc) when conveying the actions. Warning: some of the behaviors are applied iff the camera style and the conveyed action allows it.

double intensity () const

returns the current intensity value

bool isAllowed (std::string style_name)

Whether the camera is allowed to have a particular behavior (panning, traveling, zooming, etc) when conveying the actions (.

bool makeCut (double duration) const

Whether or not the camera should make a cut at this frame or not, by taking into account a current shot duration, and a probability function on the shot duration interval.

void setIntensity (double val)

Changes the intensity value.

Style ()

Constructor.

void updateIntensity (double dt)

Updates the intensity value wrt to the increase/decrease factor.

Public Attributes

Atmosphere atmosphere
double cutIntervalLength
std::string dimension
Semantic::ActorConfiguration dimension_actors
double dutch_angle
static const int DY_POSITION = DY_ORIENTATION<<1
static const int DY_TRANSITION = DY_POSITION<<1
Dynamicity dynamicity

dynamicity

Intensity m_intensity_behaviour
static std::string PANNING_BEHAVIOR = "PANNING BEHAVIOR"
Base::Interval< double > shotDurationInterval
static std::string TRAVELING_BEHAVIOR = "TRAVELING BEHAVIOR"
Base::Interval< double > visibilityInterval

Static Public Attributes

static const std::string DEFAULT_DIMENSION = "default"
static const int DY_ORIENTATION = 1
static std::string HAND_HELD_BEHAVIOR = "ON THE SHOULDER BEHAVIOR"

Protected Attributes

std::set< std::string > m_allowed_styles

Detailed Description

This class represents a cinematic style, containing high-level style indicators such as the camera dynamicity, the cinematic dimension, the dramatic tension (called intensity), the pacing or the visibility of key subjects.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Member Enumeration Documentation

enum Editing::Style::Atmosphere

Enumerator:

Stability
Instability

enum Editing::Style::Dynamicity

Enumerator:

NO_DYNAMICITY
ORIENTATION
POSITION_ORIENTATION
FULL

enum Editing::Style::Intensity

Intensity values

Enumerator:

NO_INTENSITY
INCREASE
DECREASE

Editing::Idiom Class Reference

Represents the series of characteristic viewpoints related to conveying a given action.
`#include <Idiom.h>`

Public Member Functions

void addViewpoint (Editing::Viewpoint *v)
add a viewpoint in the progression of the idiom

Editing::Viewpoint * getViewpointByIndex (int n)
get a viewpoint from its index

Editing::Viewpoint * getViewpointByPercent (double percent)
get a viewpoint from the percentage of progress

Idiom (std::string name)
Constructor.

std::string name ()
Get the name of the idiom.

int viewpointSize ()
returns the size of the set of used characteristic viewpoints

~Idiom ()
Destructor.

Static Public Member Functions

static void clearAll ()
clear all the static elements

static Idiom * getIdiom (std::string name)
get the idiom from its name

static Idiom * loadFromXML (TiXmlElement *)
Loads an idiom from an XML file.

Protected Attributes

std::string m_name
std::vector< Editing::Viewpoint * > m_viewpoints

Static Protected Attributes

static std::map< std::string, Idiom * > m_all_idioms

Detailed Description

Represents the series of characteristic viewpoints related to conveying a given action.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Editing::Viewpoint Class Reference

Represents a characteristic viewpoint of the cinematography
`#include <Viewpoint.h>`

Public Member Functions

bool operator!= (const Viewpoint &vp) const
check the inequality with a given viewpoint

bool operator< (const Viewpoint &vp) const
check whether the viewpoint should be placed before or after a given viewpoint in a sorted list

bool operator== (const Viewpoint &vp) const
Check teh equality with a given viewpoint.

std::string toString ()
get a representation of the viewpoint as a string

Viewpoint ()
Constructor.

~Viewpoint ()
Destructor.

Static Public Member Functions

static framing & getFrameComposition (const Viewpoint &vp, const std::string &dimension)
Search the frame composition to apply for a given viewpoint, wrt a given cinematic dimension.

static void loadFrameCompositionsFromXML (TiXmlElement *pElem)
Loads the set of frame compositions from an XML file.

static Viewpoint * loadFromXML (TiXmlElement *vpElem)
Loads a viepoint from an XML file.

Public Attributes

Semantic::VolumeParameters::ActorSide actor
bool change_loi_side
Semantic::VolumeParameters::Distance shot
Semantic::VolumeParameters::Type type

Protected Types

typedef std::map< std::string, framing > FCMap

Static Protected Attributes

static std::map< Viewpoint, FCMap > m_compositions

Detailed Description

This class represents a characteristic viewpoint of the cinematography. To each characteristic viewpoint is associated a characteristic framing for each cinematic dimension (i.e. one framing for conveying dominance, and one for conveying affinity).

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Editing::FilteringProcess Class Reference

Process of filtering and annotating a set of Director volumes wrt a set of cinematic rules.
`#include <FilteringProcess.h>`

Public Types

enum Directive { CUT, CUT_EXTREME_LONG, TRANSITION }

Public Member Functions

Filters::DVSortedList * filter (Directive d=CUT, int nbResults=1)
FilteringProcess (Editing::Editor &ed)
CostFunction * getCostFunction () const

Acces the cost function used to order the tuples.

void setEvent (Story::Event *e)

Changes the conveyed event, this will modify the behavior of the filters wrt director volumes.

~FilteringProcess ()

Protected Attributes

CostFunction * m_cost_function
Director::Map * m_director_map
Editing::Editor & m_editor
Story::Event * m_event

Detailed Description

This represents the process of filtering and annotating a set of Director Volumes wrt a set of cinematic rules. The filtering is different according to the directive that is specified (Cut, make a continuous Transition or Cut to an Extreme Long shot), which will influence the series of filters that are applied on Director Volumes.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Member Enumeration Documentation

enum Editing::FilteringProcess::Directive

Enumerator:

CUT
CUT_EXTREME_LONG
TRANSITION

Editing::ConfiguringProcess Class Reference

Process that fix the parameters of a camera wrt a given 2D5 director volume and a set of constraints on the camera Mimics the behavior to the photography director and the camera operator.

`#include <ConfiguringProcess.h>`

Public Types

enum ProcessMode { PM_NONE, PM_TRANSITION, PM_TRAVELING }

Public Member Functions

void configureCameraInVolume (Camera &, const Semantic::VolumeParameters &, Base::ConvexCell *, std::string &, Semantic::ActorConfiguration &)
Configure the position/orientation of a camera inside a given volume.

void configureCameraInVolume (Camera &, const Semantic::VolumeParameters &, Base::ConvexCell *, Semantic::ActorConfiguration &, std::vector< Vector2d > &)
Configure the position/orientation of a camera inside a given volume.

void configureCut (Camera &, Filters::DVTuple &t)
configures the camera parameters to make a cut from the last camera position to the given DV, orienting the camera to match the cinematographic needs

void configureTransition (Filters::DVSortedList &l)
configures the camera wrt to a smooth transition

ConfiguringProcess (Editor &)
Constructor.

ProcessMode mode ()
returns the camera motion mode

void reconfigure (Camera &, double dt=0)
reconfigures the camera parameters during a shot by using the given information: the camera parameters, and may be the given time increase the style of the editor, as a base to make variations in the camera parameters evolution

void setMode (ProcessMode m)
sets the camera motion mode (following an actor, making a transition or none)

~ConfiguringProcess ()
Destructor.

Static Public Member Functions

static void ConfiguringProcess::clearAll ()

clears all the structure of the configuring processes (use only when shutting down)

Protected Attributes

MovableVector< 3, REAL > m_camera_HH_location

The camera location variation for the "hand held" behavior.

MovableVector< 3, RADIAN > m_camera_HH_orientation

The camera orientation variation for the "hand held" behavior.

Vector3d m_camera_HH_target_location

The camera target location variation for the "hand held" behavior.

Vector3d m_camera_HH_target_orientation

The camera target orientation variation for the "hand held" behavior.

MovableVector< 3, RADIAN > m_camera_panning

The camera location wrt the "panning" behavior.

MovableVector< 3, REAL > m_camera_traveling

The camera location wrt the "traveling" behavior.

Editor & m_editor

the associated editor

FrameComposition * m_frameComp

The module that process the frame composition within a volume.

ProcessMode m_mode

The processing mode.

std::set< Director::Volume2d5 * > m_transitionTargets

The set of targets when initiating a transition.

Detailed Description

This is the process that fix the parameters of a camera wrt a given 2D5 director volume and a set of constraints on the camera. It mimics the behavior to the photography director and the camera operator.

This process use the Frame Composition class to process the best camera configuration within a given Director Volume or reconfigure the camera orientation to maintain that composition while moving the camera.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Editing::FrameComposition Class Reference

Structure for processing the frame composition of key elements on screen.

```
#include <FrameComposition.h>
```

Public Member Functions

void findBestCamera (Base::ConvexCell *cell, Vector3d &position, Vector3d &direction)
Search the given cell for the best camera position and direction to fulfill the screen position of the characters.

FrameComposition (const Editing::Camera &c, Semantic::ActorConfiguration actorsConfig, std::vector< Vector2d > screenPosition)
Constructor.

Vector3d getDirectionForSatisfaction (const Vector3d &position) const
Compute the direction of the camera wrt the satisfaction of the first actor.

void updateCharactersPosition ()
Updates the position of each character by retrieving them from the configuration.

Protected Member Functions

double getPointSatisfaction (const Vector3d &position) const
double getSatisfaction (const Vector3d &position, const Vector3d &direction) const
double searchNeighbour (Base::ConvexCell *cell, Vector3d &searchPoint, Vector3d &d)
TODO: Comment this function.

Protected Attributes

Semantic::ActorConfiguration m_actor_configuration
The name of the characters.

const Editing::Camera & m_camera
the camera that frame the scene

bool m_reversed
whether the composition value should be reversed (case of A and B in a reversed position)

std::vector< Vector3d > m_scenePosition
The 2D scene position of the characters.

std::vector< Vector2d > m_screenPosition
The intended screen x-position of the first character.

double m_tanFovX_2
The x-fov of the camera.

double m_tanFovY_2

Detailed Description

This describes the structure used for processing the frame composition of key elements on screen.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Filters::Annotation Class Reference

`#include <IFilter.h>`

Public Member Functions

Annotation ()
Constructor.

void erase (std::string s)
erase the annotation associated to a specific attribute

int get (std::string s) const
get an annotation associated to a specific attribute

void set (std::string s, int a)
set an annotation associated to a specific attribute

Protected Attributes

std::map< std::string, int > m_annotations

Filters::DVTuple Class Reference

Represents a couple of a Director Volume and the set of Annotations added by filters.
`#include <IFilter.h>`

Public Member Functions

Annotation *& annotation ()
get the set of annotations of the tuple

Director::Volume2d5 *& dv ()
get the director volume of the tuple

DVTuple (std::pair< Director::Volume2d5 *, Annotation * > p)
Constructor.

DVTuple (Director::Volume2d5 *dv, Annotation *a)
Constructor.

~DVTuple ()
Destructor.

Detailed Description

DVList tuple : a director volume + a set of annotations

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

4/2/2010

Filters::DVList Class Reference

Represents a non sorted list of DVTuple elements.
`#include <IFilter.h>`

Public Member Functions

void deleteAll ()
delete all the tuples

DVList ()
Constructor.

DVList (std::set< DVTuple * > &s)
Constructor.

Filters::IFilter Class Reference

Represents a filter that works on a given criteria to annotate and/or prune a set of Director Volumes.
`#include <IFilter.h>`

Public Member Functions

virtual double cost (DVTuple *) const =0
Get the cost of the tuple wrt the filter.

virtual DVList * filter (DVList *const) const =0
filter a list of DV tuples wrt a given characteristic

virtual std::string type () const

Detailed Description

Represents a filter that work on a given criteria to annotate and prune some director volumes. The implemented filters are: AreaFilter, CoherencyFilter, CurrentSVFilter, DistanceFilter, IdiomFilter; IntensityFilter, LoAFilter, LoIFilter, ThirtyDegreeFilter, VisibilityFilter.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/14

Director::Map Class Reference

Describes a map merging both visibility and semantic information on a configuration of actors.
`#include <Map.h>`

Public Member Functions

void allVolumes2d5 (std::vector< BSP::Leaf * > &l) const
returns all the BSP leaves (ie director volumes) from all the BSP trees

void displayOgre (Demo &out) const
displays the map into Ogre

std::vector< Semantic::Actor * > getActors () const
returns the configuration of actors

Planning::Roadmap * getRoadmap () const
return the roadmap for path planning

Semantic::Tree * getSemanticTree () const
returns the semantic BSP tree

void print (Printer::SVG::OutSVG &out) const
prints the map into an SVG file

BSP::Tree * tree (Topology::CellBase::CellId cellid) const
returns the corresponding BSP tree

Semantic::Volume * volume (Vector2d &p) const
return the semantic volume that corresponds to a viewpoint

Volume2d5 * volume2d5 (Vector2d &p) const
return the director volume that corresponds to a given viewpoint

Static Public Member Functions

static void clearAll ()
clear all created Maps

static Map * getSingleton (Editing::Editor &creator, std::vector< Semantic::Actor * > &a, bool getVisibility=true, bool semantic=true)
get a unique Map by its parameters a Map is created iff no equivalent Map already exists

static Map * getSingleton (Editing::Editor &creator, Semantic::ActorConfiguration &c, bool getVisibility=true, bool semantic=true)
get a unique Map by its parameters a Map is created iff no equivalent Map already exists

Protected Attributes

std::vector< Semantic::Actor * > m_actors
Editing::Editor & m_creator
std::map< std::string, Visibility::Map * > m_maps
Planning::Roadmap * m_roadmap
bool m_semantic
Semantic::Tree * m_SVTree
BSP::TreeMap m_trees
bool m_visibility

Static Protected Attributes

static std::map< std::string, Map * > ALL_MAPS

Detailed Description

This class describes a map merging both visibility and semantic information on a configuration of actors. It contains the whole set of 2D5 Director Volumes on the given configuration of actors.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/07

Director::Volume2d5 Class Reference

Describes a 2D5 volume merging both visibility and semantic information.
`#include <Volume2d5.h>`

Public Member Functions

void _addTag (Planning::TagId id)
void _categorize (Semantic::Actor *act, Visibility::Volume2d5 *l)
categorizes wrt the region for the given actor
void _categorize (Semantic::Volume *v)
categorizes wrt the SV
Planning::TagIdList _getTagList ()
bool checkConfigurationVisibility (Semantic::ActorConfiguration &c, double threshold) const
Check whether a sub-set of actor is visible enough (ie v >= threshold) in the volume.
void displayOgre (Demo &out)
double * getAvgVisibility (Semantic::Actor *act=NULL) const
Process the average visibility in the volume The visibility is related to a given actor iff act != NULL or to whole configuration of actors elsewise.
Map * getCreator ()
Returns the parent map.
const Semantic::Volume * getSV () const
returns the Semantic information
double * getVisibility (Vector2d &p, Semantic::Actor *act=NULL) const
Process the visibility from a given viewpoint The visibility is related to a given actor iff act != NULL or to whole configuration of actors elsewise.
double getVisibility (Semantic::ActorConfiguration &c) const
Process the average visibility of na given configuration of actors in the volume.
double maxVisibility (Semantic::Actor *act=NULL) const
returns the maximum visibility value in the volume The visibility is related to a given actor iff act != NULL or to whole configuration of actors elsewise
double minVisibility (Semantic::Actor *act=NULL) const

returns the minimum visibility value in the volume The visibility is related to a given actor iff act != NULL or to whole configuration of actors elsewise

void print (Printer::SVG::OutSVG &out)
Prints into a SVG file.

virtual BSP::Node * subdivide (BSP::Plane *)
Returns the new child node on which his parent node should point to.

Volume2d5 (Volume2d5 &v)
Copy constructor.

Volume2d5 ()
Default constructor (used only for serialization)

Volume2d5 (Map *creator, Base::ConvexCell *cell)
Constructor.

virtual ~Volume2d5 ()
Destructor.

Protected Member Functions

Printer::SVG::Color * color ()

Protected Attributes

Map * m_creator
std::map< std::string, Visibility::Volume2d5 * > m_leaves
Semantic::Volume * m_SV
Planning::TagIdList m_tagList

Detailed Description

This class describes a 2D5 volume merging both visibility and semantic information about a given configuration of actors.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/07

Planning::Roadmap Class Reference

represent the camera motion planning graph
`#include <Roadmap.h>`

Public Member Functions

TagId _addTag (TopoExtension::Segment &s)
NodeIdList _getNodeIdList (Topology::Portal *p)

NodeInfo * _getNodeInfo (RMType::NodeId)
void _processRoadmap ()
void _splitTag (TagId id, TopoExtension::Line const &l, TagIdList &posList, TagIdList &negList)
virtual void displayOgre (Demo &out)
Displays the roadmap into Ogre.
::Graph::GenericGraph< RMType::NodeId, RMType::ArcId, NodeInfo *, ArcInfo * > & operator() ()
Get the planning graph.
void optimizePath (std::vector< Vector2d > &path, double dist)
Optimizes a processed camera path by using the environment topology.
void path (Vector2d origin, std::set< Director::Volume2d5 * > dest, Editing::Style &style, std::vector< Vector2d > &path)
Processes the camera path.
Roadmap::Roadmap (Director::Map *creator, double d=WAYPOINT_DENSITY)
Constructor.
~Roadmap ()
Destructor.

Static Public Attributes

static int OPTIMIZE_NB_PASS = 1
static double WAYPOINT_DENSITY = 1

Protected Member Functions

bool _freeFromObstacle (Geo2d::Triangle t) const

Protected Attributes

std::set< ArcInfo * > m_arcs
Director::Map * m_creator
::Graph::FloodFill< double, RMType::NodeId, RMType::ArcId, NodeInfo *, ArcInfo *, ArcValueFunction > *
m_floodfill
::Graph::GenericGraph< Planning::RMType::NodeId, Planning::RMType::ArcId, NodeInfo *, ArcInfo * > *
m_graph
Type::ExtendedVector< NodeIdList, TagId > m_nodes
std::map< Topology::PortalBase::PortalId, Tag > m_portalTags
std::map<TopoExtension::Segment, RMNodeIdSet > m_nodes;
bool m_processed
whether this roadmap has already been processed
std::map< TagId, TopoExtension::Line > m_split_line
std::map< TagId, TagIdPair > m_tag_split
Type::ExtendedVector< Tag, TagId > m_tags
double m_wp_density

Detailed Description

This class represents the camera motion planning graph, which is built over a Director Map and is used for planning any camera motion between two successive viewpoints.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Semantic::Actor Class Reference

Represents the minimal information about an actor needed to create semantic and visibility volumes.

`#include <Actor.h>`

Public Member Functions

Actor (std::vector< Vector2d > cloud, double visibility_update_time=0)
Constructor.

virtual void deserialize (Raw::InRawFile &in)
Deserialization of the object inheriting from this class.

const Geo3d::Surface2d5::Face2dId & getHeuristic () const
Returns the search heuristic of the actor location wrt to topological cells.

std::string getName () const
Returns the name of the actor.

double getOrientation () const
Returns the orientation of the actor (in radian)

const Vector2d & getPosition () const
Returns the position of the actor in 2D.

const Vector2d & getSpeed () const
Returns the current speed vector of the actor.

const Topology::CellIdSet & in () const
returns the set of topological cells teh actor is in

bool isIn (Topology::CellBase::CellId const &cid) const
Check whether the actor is in a given topologiocal cell.

Visibility::Map * map (Semantic::Tree *stree=NULL)
Returns the last updated visibility information on the actor.

Actor * photograph () const
Make a photograph of teh actor, ie copy its properties (position, oreinattion, etc) into a new actor.

void print (Printer::SVG::OutSVG &out)
Draws the actor in an SVG file.

void processIn ()
mets à jour la liste des cellules d'appartenance

double radius () const
Returns the radius of the convex shape representing the actor.

void reset ()
resets the properties of the actor

TopoExtension::Segment segment2d (int i) const
Returns the ith edge of the actor.

virtual void serialize (Raw::OutRawFile &out) const

Serialization of the object inheriting from this class.

void setName (std::string n)

Changes the name of the actor.

void setVisibilityUpdateTime (double t)

sets the visibility update time

void update ()

Update the properties of the actor.

Vector2d vertex (int i) const

Returns the ith vertex of teh convex shape of the actor.

virtual ~Actor ()

Destructor.

Static Public Member Functions

static void clearAll ()

Clean the list of referenced actors.

static Actor * getActorByIndex (const int idx)

Search an actor by its index in the list.

static Actor * getActorByName (const std::string &name)

Search an actor by its name.

static int nbActors ()

returns the total number of referenced actors

static void reference (const std::string &name, Actor *a)

Save a reference onto an actor.

Protected Member Functions

void initThread ()

Protected Attributes

Geo3d::Surface2d5::Face2dId m_heuristic
std::set< Topology::CellBase::CellId > m_in
Visibility::Map * m_map
MapThread * m_map_thread
std::string m_name
double m_orientation
Vector2d m_position
Vector2d m_speed
double m_visibility_update_time

Static Protected Attributes

static std::map< std::string, Actor * > m_all_actors

Detailed Description

This class represents the minimal information about an actor needed to create semantic and visibility volumes (3D position, orientation) and make the link between an actor of the demonstration and the information that is needed about actors for the camera control system.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Semantic::ActorConfiguration Class Reference

Represents a configuration of actors relative to a given event.
`#include <ActorConfiguration.h>`

Public Member Functions

Actor * actor (int i) const
get a actor from its index in the configuration

ActorConfiguration ()
Constructor.

ActorConfiguration (Actor *A, Actor *B=NULL, Actor *C=NULL)
Constructor.

ActorConfiguration (std::string A, std::string B="", std::string C="")
Constructor.

ActorConfiguration (std::vector< std::string > act)
Constructor.

ActorConfiguration (std::vector< Actor * > act)
Constructor.

std::vector< Actor * > actors () const
get the list of actors composing the configuration

TopoExtension::Line getLOA (int i) const
get a line of action of the configuration by its index

TopoExtension::Line getLOI (int i=0) const
get a line of interest of the configuration by its index

bool insersect (Semantic::ActorConfiguration &c)
Check whether there are shared actors with a given configuration.

int nbLOA () const
get the number of LoA

int nbLOI () const
get the number of LoI

bool operator!= (ActorConfiguration const &c) const
Check whether a configuration is different from the current configuration.

ActorConfiguration operator* (const ActorConfiguration &) const
Process the intersection between two configurations.

bool operator< (ActorConfiguration const &c) const
whether a given configuration should be placed before the given configuration or not in a sorted list

bool operator== (ActorConfiguration const &c) const
Check whether a configuration is equal to the current configuration.

std::string operator[] (int i) const
get the name of the actor i

void print (Printer::SVG::OutSVG &out)
draws the configuration into an SVG file

ActorConfiguration reversed () const
Reverse the configuartion of actors.

int size () const
return the number of actors composing the configuration

~ActorConfiguration ()
Destructor.

Static Public Member Functions

static ActorConfiguration getOnScreenActors (const Editing::Camera &c)
Process the set of actors that appear on the screen.

Protected Attributes

std::vector< std::string > m_actors

Detailed Description

This class represents a configuration of actors relative to a given event
The first actor in the configuration is the actor A, the second is the actor B, etc.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

4/2/2010

Semantic::Tree Class Reference

Represents the BSP tree that contains the semantic information relative to a given configuration of actors.
`#include <Tree.h>`

Public Member Functions

void clear ()
Clean the structure.

Volume getMergedVolume (VolumeParameters &vp) const
Merge the set of volumes that share the same parameters.

TopoExtension::CellSelector * getSelector () const
returns the spatial selector associated to the tree (wrt cells intersecting the semantic volumes)

virtual void lines (std::set< TopoExtension::Line > &l) const

Returns the set of splitting lines associated to the tree.

Tree (const Editing::Camera &cam, const ActorConfiguration &c, const Editing::Style &style)
Constructor.

Volume * volume (Vector2d &p) const
returns the semantic volume that contains a 2D point

void volumes (std::vector< Semantic::Volume * > &v) const
Returns the set of semantic volumes.

~Tree ()
Destructor.

Protected Member Functions

void _AddAvoidanceOfOtherCharacters (const ActorConfiguration &c)
void _categorizeWithTags ()
void _createTreeWithOneActor (ActorConfiguration c)
void _createTreeWithTwoActors (ActorConfiguration c)
void _insertTag (Tag, Regions::IRegion *)
void _splitWith (Base::ConvexCell *)
void _splitWith (TopoExtension::Line &l)

Protected Attributes

const Editing::Camera & m_camera
std::set< Base::ConvexCell * > m_cells
TopoExtension::CellSelector * m_selector
const Editing::Style & m_style
std::vector< std::pair< Tag, Regions::IRegion * > > m_tags

Static Protected Attributes

static Volume OUT_VOLUME

Detailed Description

This class represents the BSP tree that contains the semantic information relative to a given configuration of actors.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Semantic::Volume Class Reference

Represent a semantic volume, ie a characteristic viewpoint, represented as a 2D cell
`#include <Volume.h>`

Public Member Functions

virtual VolumeParameters & parameters ()
Returns the parameters.

virtual const VolumeParameters & parameters () const
Returns the parameters.

virtual BSP::Node * subdivide (BSP::Plane *)
subdivide the volume with a given plane

Volume ()
Constructor.

Volume (const Volume &)
Copy constructor.

Volume (const VolumeParameters &)
Constructor.

TiXmlElement * Volume::getAsXML (bool cellCoordinates=false) const
Creates and returns an XML node that carry the information about the volume.

virtual ~Volume ()
Destructor.

Protected Types

typedef std::vector< Director::Volume2d5 * > DVSet

Protected Attributes

VolumeParameters m_parameters

Detailed Description

This class represents the semantic information related to a characteristic viewpoint, with information about the side of the Volume wrt the Line of Interest.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/13

Semantic::VolumeParameters Class Reference

Represents the parameters of a semantic volume (Type of shot, distance from actors, side wrt the line of interest (LOI), side wrt actors)

```
#include <Volume.h>
```

Public Types

enum ActorSide { A_SIDE = 0, B_SIDE = 1, ANY_ACTOR_SIDE = -1 }

enum Distance { ANY_SHOT = 0, SUBJECTIVE = 1, CLOSEUP = 2, CLOSE = 3, MEDIUM_CLOSEUP = 4, LONG = 5, EXTREME_LONG = 6 }
typedef Geo2d::Segment::Side LoISide
enum Type { ANY_TYPE = 0, EXTERNAL = 1, INTERNAL = 2, PARALLEL = 3, FRONT = 4, FRONT_34 = 5, PROFILE = 6, BACK_34 = 7, BACK = 8, OUTWARDS = 9, PROHIBITED = 10 }

Member Enumeration Documentation

enum Semantic::VolumeParameters::ActorSide

Enumerator:

A_SIDE
B_SIDE
ANY_ACTOR_SIDE

enum Semantic::VolumeParameters::Distance

Enumerator:

ANY_SHOT
SUBJECTIVE
CLOSEUP
CLOSE
MEDIUM_CLOSEUP
LONG
EXTREME_LONG

enum Semantic::VolumeParameters::Type

Enumerator:

ANY_TYPE
EXTERNAL
INTERNAL
PARALLEL
FRONT
FRONT_34
PROFILE
BACK_34
BACK
OUTWARDS
PROHIBITED

Visibility::Map Class Reference

Visibility map wrt an actor, computed on the whole environment.
`#include <Map.h>`

Public Member Functions

const Prism * _prism (VConstraint const &p) const
Access the prism associated to a visibility constraint.

BSP::Tree * _tree (Topology::CellBase::CellId const &cellid) const
Access the BSP tree associated to a topological cell.

void computeVisibleSet (Semantic::Tree *stree)
Calculates the visible set.

void displayOgre (Demo &out)
Displays all the visibility volumes into Ogre.

double lastUpdate () const
Access the last update time.

Map (Semantic::Actor *act)
Constructor.

bool needUpdate (double update_time=0) const
Tests the needs to a visibility update.

void print (Printer::SVG::OutSVG &out)
Draws all the visibility volumes into an SVG file.

Volume2d5 * volume2d5 (Vector2d const &p) const
Access the visibility volume thats containsn a 2d point.

~Map ()
Destructor.

Static Public Attributes

static bool CHECK_SV_INTERSECTION = false

Protected Types

typedef std::pair< Topology::CellBase::CellId, Topology::PortalBase::PortalId > CPPair
map of additional stabbing primitives with some additional information (not propagated)

typedef std::map< VConstraint, Prism *, VConstraintSorter > PrismMap
BSP trees each wrt a topological cell.

Protected Attributes

Semantic::Actor * m_actor

std::vector< Vector2d > m_actor_hull
set of all valid topological cells

std::string m_actor_name

std::multimap< VConstraint, VConstraint > m_inclusions

double m_last_update

std::set< Prism * > m_prismList
each prism is relative to a visibility constraint

PrismMap m_prisms

BSP::TreeMap m_trees

std::set< std::pair< VConstraint, CPPair > > m_visited

std::map< Topology::CellBase::CellId, VPlaneMap > m_VPlaneLists
maximum time before an update is needed

VPlaneMultimap m_VPlaneMultiMap
the actor

VPlaneMultimap m_VPlaneMultiMap_additional
map of stabbing primitives with some additional information

Topology::CellIdSet valid_cells
last update date

Static Protected Attributes

static const double DEFAULT_UPDATE_TIME = 0.

Detailed Description

Visibility map wrt an actor, computed on the whole environment. It is used to access the Visibility Volumes, that is viewpoints from where a given actor is totally visible, totally occluded or partially visible (and in this case process the visible portion of that actor).

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

11/30/2009

Visibility::Volume2d5 Class Reference

Represents a visibility volume related to an actor.

```
#include <Volume2d5.h>
```

Classes

class VConstrSorter

Public Member Functions

void add (VConstraint c)
Adds a visibility constraint to the volume.

void add (std::set< VConstraint > c)
Adds a set of visibility constraints to the volume.

Categorization categorization ()
Returns the categorization type of the volume.

virtual void displayOgre (Demo &out)
Displays the volume into Ogre.

double * getAvgVisibility ()
Process the average visibility value.

double * getVisibility (Vector2d const &)
Process the visibility value from a given viewpoint in the volume.

virtual void print (Printer::SVG::OutSVG &out)
Draws the volume into an SVG file.

void setCellId (Topology::CellBase::CellId id)
Sets the id of the parent topological cell.

void setCreator (Map *)
Sets the parent visibility map.

virtual BSP::Node * subdivide (BSP::Plane *)
Subdivides the volume according to a stabbing plane.
Volume2d5 (Volume2d5 &)
Copy constructor.
Volume2d5 (Base::ConvexCell *cell)
Constructor.
virtual ~Volume2d5 ()
Destructor.

Protected Member Functions

void categorize ()
void categorize (Categorization c)

Protected Attributes

Categorization m_categorization
identifier of the parent topological cell
std::set< VConstraint > m_constraints
Map * m_creator
Topology::CellBase::CellId m_parent_cid
parent visibility map
TagList m_tagList

Detailed Description

This class represents a 2D5 visibility volume related to an actor as a 2D5 Cell, which is tagged with visibility information (total visibility, total occlusion or partial visibility).

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/14

Topology::Graph Class Reference

Represents a topological Cell and Portal Graph (CPG)
`#include <Graph.h>`

Public Types

enum EnvType { Camera, Actor }
Type of environment: either the graph is used for actors or for cameras.

Public Member Functions

Cell * cell (CellBase::CellId const &cid) const

Access a cell by its id.

Cell * cell (int i) const
Access a cell by its index.

CellBase::CellId cellId (Vector2d const &pt, Geo2d::TrianglePartitionBase::FaceId &heuristic=Geo3d::Surface2d5::Face2dId(0)) const
Access the id of the cell that contains a 2D point or -1 if not found.

CellBase::CellId cellId (Geo2d::TrianglePartitionBase::FaceId const &fid) const
Access the id of the cell that includes a given 2D face.

const CellIdSet & cellIdSet (int subgraph_idx) const
Access the set of cells that are in a given connected subgraph.

CellIdPair cells (PortalBase::PortalId const &pid) const
Acess the pair of cell connected by a given portal.

int cellSize () const
Access the total number of cells.

virtual void deserialize (Raw::InRawFile &in)
Deserialization of the object inheriting from this class.

void displayOgre (Demo &out, int sg=-1)
Displays the graph into Ogre.

Pair< double > floor_ceil (Vector2d const &p) const
Access the floor height and the height under ceil at the 2D point p.

bool freeFromObstacle (const Geo2d::Segment &s) const
Tests if a segment is free from obstacles in the environment.

bool freeFromObstacle (const Geo2d::Triangle &t) const
Tests if a triangle is free from obstacles in the environment.

Graph ()
min/max height under ceil, floor height

bool intersectWall (TopoExtension::Segment const &s, const CellBase::CellId &origin) const
Tests the intersection between a segment starting form a given topological cell and a wall in the environment.

CellIdSet neighbours (CellBase::CellId const &cid) const
Access the set of neighbours cells of a given cell.

::Graph::GenericGraph< CellBase::CellId, PortalBase::PortalId, Cell *, Portal * > & operator() ()
Returns the basical graph structure.

Portal * portal (PortalBase::PortalId const &pid) const
Access a portal by its id.

Portal * portal (int i) const
Access a portal by its index.

int portalSize () const
Access teh total number of portals.

void print (Printer::SVG::OutSVG &out) const
Draw the graph into an SVG file.

virtual void serialize (Raw::OutRawFile &out) const
Serialization of the object inheriting from this class.

void setFloorHeight (double f)
Changes the default floor height.

void setMaximumHeight (double h)
Changes the maximum height under ceil.

void setMinimumHeight (double h)

Changes the minimum height under ceil.

const std::vector< TopoExtension::Segment > & sharing (Vector2d const &p) const
Acces the set of segments of the initial decomposition that share a given vertex.

int subgraph (CellBase::CellId const &cid) const
Access the index of the subgraph that contains a given cell.

Static Public Member Functions

static FloorId getCurrentFloor ()
Returns the id of the considered floor.

static Geo3d::SimpleEnvironment * getEnvironment (EnvType et=Camera)
Access the environment structure corresponding a given type.

static FloorId getNbFloors ()
Returns the total number of floors.

static Graph & getSingleton (FloorId floor=FLOOR)
Returns a reference to the graph associated to a given floor.

static Graph * getSingletonPtr (FloorId floor=FLOOR)
Returns a reference to the graph associated to a given floor.

static void load (std::string camera_env, std::string actor_env)
Loads the whole structure from files.

static void save ()
Saves the graph in the defined file.

static void setFloor (FloorId new_floor)
Changes the current floor.

static void unload ()
Unload the whole structure.

Protected Attributes

static Geo3d::SimpleEnvironment * ACTOR_ENV3D = NULL
std::vector< Topology::Cell * > m_cells
Geo3d::TopologicalViewBase::SurfaceId m_floor
double m_floor_height
::Graph::GenericGraph< CellBase::CellId, PortalBase::PortalId, Cell *, Portal * > m_graph
double m_maximum_height
double m_minimum_height
std::map< Geo3d::Surface2d5::Face2dId, Topology::CellBase::CellId > m_partOf
std::vector< Topology::Portal * > m_portals
std::map< Vector2d, std::vector< TopoExtension::Segment > > m_shared
std::vector< CellIdSet > m_subgraphs

Static Protected Attributes

static Geo3d::SimpleEnvironment * CAMERA_ENV3D = NULL
static std::string FILE
static FloorId FLOOR
static std::vector< Graph * > GRAPH_BY_FLOOR

Detailed Description

This class represents a topological Cell and Portal Graph (CPG) which is associated to a given floor. When loading the CPG structure from a file, a CPG is loaded for each floor.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/14

Member Enumeration Documentation

enum Topology::Graph::EnvType

Type of environment: either the graph is used for actors or for cameras.

Enumerator:

Camera
Actor

Topology::Cell Class Reference

represents a topological cell, used in the spatial partitioning of the environment
`#include <Cell.h>`

Public Member Functions

void associate (Portal *p)
Associate az portal to the cell.

Cell ()
number of portals sharing a given vertex

Cell (std::vector< TopoExtension::Segment > base)
Constructor.

Cell (std::vector< Vector2d > &hull)
Constructor.

const CellBase::CellId & cid () const
Returns the id of the cell.

virtual void deserialize (Raw::InRawFile &in)
Deserialization of the object inheriting from this class.

bool intersectWall (TopoExtension::Segment const &s) const
Tests the intersection of a segment with a wall of the topological cell (ie a edge of the cell)

bool isPortal (int i) const
Tets if the ith edge is a portal.

void operator= (Cell &c)
Copy a cell.

bool operator== (const Cell &c) const
Tests the equality with an other topological cell.

const Portal * portal (int i) const
Returns the ith portal associated to the cell.

int portalSize () const
Returns the number of portals associated to the cell.

virtual void serialize (Raw::OutRawFile &out) const
Serialization of the object inheriting from this class.

int sharingPortals (Vector2d p) const
Returns the number of portals that share a given vertex.

~Cell ()
Destructor.

Static Public Member Functions

static std::pair< Cell *, bool > merge (Cell const *c1, Cell const *c2, Portal const *p)
Merge two cells according to given portal and tests if the merged cell is convex.

Static Public Attributes

static const char * TYPE = "Topology::Cell"

Protected Member Functions

bool isConvex () const
void simplify ()

Protected Attributes

CellBase::CellId m_id
std::vector< Portal * > portals
std::map< Vector2d, unsigned int > sharing

Friends

class Graph

Detailed Description

This class represents a topological (convex) cell, used in the spatial partitioning of the environment, to support the visibility computation, and are connected together by topological portals. A topological cell represents a convex 2D5 free space (i.e. free-from-obstacle) for placing or moving the camera.

Author:

IRIS NoE - WP5 - Newcastle University / INRIA

Date:

2010/12/14

Topology::Portal Class Reference

represents a topological portal, used in the spatial partitioning of the environment
`#include <Portal.h>`

Public Member Functions

Vector2d center () const
Process the center of the portal.

void deserialize (Raw::InRawFile &in)

void displayOgre (Demo &out, Printer::SVG::Color &col=Color::RED)
Displays the portal into Ogre.

bool operator== (Portal &p) const
Tests the equality with another portal.

const Vector2d & operator[] (int i) const
Access a vertex of the portal.

const PortalBase::PortalId & pid () const
Access the id of the portal.

Portal (TopoExtension::Segment s)
Constructor.

Portal ()
Degault constructor (used for serialization)

void print (Printer::SVG::OutSVG &out)
Draws the portal in an SVG file.

const TopoExtension::Segment segment () const
Access the segment corresponding to the portal.

void serialize (Raw::OutRawFile &out) const

const Vector2d & vertex (int i) const
Access a vertex of the portal.

~Portal ()
Destructor.

Protected Attributes

PortalBase::PortalId m_id

TopoExtension::Segment m_segment

Friends

class Graph

Detailed Description

This class represents a topological portal, used in the spatial partitioning of the environment to connect two topological cells, and represent the free-from-obstacle ways for moving the camera through the whole environment.

Date:

2010/12/14

5. Conclusion

This document has described into details the API of Interactive Cinematography v1.0. This software component enables the efficient computation of viewpoints in an interactive 3D environment, given its topological representation, the characters motions, actions occurring in real-time, and directorial style parameters. The process selects the most appropriate viewpoint for a given action, performs cuts when constrained by visibility, pacing or occurrence of new actions, and performs smooth transitions between different viewpoints. As such, the component provides an easy and high-level interface to control the camera in the context of Interactive Virtual Storytelling.

At the simplest, the Interactive Cinematography component can be integrated with 10 lines of code and configured through a collection of configuration files. The component can be seamlessly integrated in any Interactive Storytelling context, as long as we specify the topological representation, the events, and the motions of the characters. This documents details an example of integration.

Furthermore, the API offers all the entry points to manipulate in details the way the Director Volumes are computed, the way the cuts are performed, the way the shots are composed inside Director Volume and the way paths are planned between regions of the environment.

For all these reasons, our contribution represents a solid foundation both for exploring intricate relations between narrative and presentation in the context of interactive storytelling, and a great research tool to explore the use of new models and techniques to automate cameras in interactive contexts.