Preprocessed Visibility Short Manual

Jiří Bittner, Oliver Mattausch, Michael Wimmer
Institute of Computer Graphics and Algorithms, Vienna University of Technology

February 9, 2007

Contents

1 Introduction

This manual provides a guide to apply preprocessed visibility in a particular target engine. It will start with instruction for building the preprocessor, describes the integration of the preprocessor and the generated visibility solution, and last but not least the usage of the view cells within the target engine.

2 Build Project

Open the *GtpVisibility.sln* with *Visual Studio 2003* (2005 might work, but we did not test it recently). Set the build target to *Release*. Build the project *TestPreprocessor*. Now there should be a newly built executable *Preprocessor.exe* in *bin/release* and a library *Preprocessor.lib* in *lib/release*.

3 Generate Visibility Solution

All necessary scripts are located in the *Preprocessor/scripts* directory. The easiest way to use the scripts is with Cygwin. First some meaningful view cells must be generated for the current scene. The script *generate_viewcells.sh* is responsible for view cell generation. It takes the parameters from the file *generate_viewcells.env*. To specify the input scene, the parameter

```
Scene.filename
```

must be set to the file name of the new scene. Our preprocessor supports the formats obj and x3d. Important options for performance are

```
Hierarchy.Construction.samples
VspTree.Termination.maxLeaves
```

The first parameter sets the number of samples that is used for view cell construction. Less samples means faster computation, but maybe slightly less optimized view cells. The second parameter sets the maximal number of view cells.

Running the script will generate a visibility solution with empty view cells. Now we must compute visibility for these view cells. In the preprocessor script *generate_visibility.sh*, set the following parameter to the newly generated visibility solution.

```
ViewCells.filename
```

This script starts the preprocessor and generates the full visibility solution. This solution contains view cells and the corresponding Potentially Visible Set (PVS). Next we explain how this visibility solution can be used in a target engine.

4 Integration of Preprocessed Visibility

4.1 Requirements

Add the following libraries to your application.

- *Preprocessor.lib*
- *zdll.lib*
- *zziplib.lib*
- *xercesc_2.lib*
- *devil.lib*

- *glut32.lib*
- *OpenGL32.Lib*
- *glu32.lib*
- *glew32.lib*
- *glew32s.lib*

The libraries can be found in the following directories.

- *trunk/Lib/Vis/Preprocessing/lib/Release*
- *GTP/trunk/Lib/Vis/Preprocessing/src/GL*
- *NonGTP/Xerces/xercesc/lib*
- *NonGTP/Zlib/lib*

This include directory must be added to your Solution.

- *GTP/trunk/Lib/Vis/Preprocessing/src*

In order to employ the preprocessor in a target engine we must make the visibility solution (PVS data) available in the engine. For this purpose we associate the entities of the engine with the PVS entries from the visibility solution. For this purposse the user must implement a small number of interface classes of the preprocessor. We demonstrate this on a small example, which shows how to access preprocessed visibility in the popular rendering engine Ogre3D. Of course, the implementation has to be adapted to the requirements of a particular target engine.

```
// this class associates PVS entries
// with the entities of the engine.
OctreeBoundingBoxConverter bconverter(this);

// a vector of intersectables
ObjectContainer objects;

// load the view cells and their PVS
GtpVisibilityPreprocessor::ViewCellsManager *viewCellsManager =
         GtpVisibilityPreprocessor::ViewCellsManager::LoadViewCells
                                    (filename, &objects, &bconverter);
```

This piece of code is loading the view cells into the engine. Let's analyze this code. There are two constructs that need explanation, the BoundingBoxConverter and the ObjectContainer, and the view cells manager.

BoundingBoxConverter This is one of the interfaces that must be implemented In this case, we implemented an OctreeBoundingBoxConverter for the Ogre OctreeSceneManager. The bounding box converter is used to associate one or more entities (objects) in the engine with each pvs entry of the visibility solution. This is done by geometric comparison of the bounding boxes.

In the current setting we compare not for equality but for intersection. All entities of the engine intersecting a bounding box of a PVS entry are associated with this PVS entry. This means that often more than one entity in the engine will map to a particular pvs entry. This gives a slight overestimation of PVS but yields a very robust solution.

ObjectContainer The object container is basicly a vector of *Intersectable* *. It contains all static entities of the scene. A PVS entry must be derived from this class. To get access to the PVS of a view cell, the user must implement this interface as a wrapper for the entities in the particular engine.

ViewCellsManager The loading function returns a view cells.

```
static ViewCellsManager *LoadViewCells(const string &filename,
   ObjectContainer *objects,
   bool finalizeViewCells = false,
   BoundingBoxConverter *bconverter = NULL);
```

The user has to provide the filename of the visibility solution, an ObjectContainer containing all the static entities of the scene, and a bounding box converter. From now on, the view cells manager is used to access and manage the view cells. For example, it can be applied to locate the current view cell. After this step the view cells should be loaded and accessable in the engine.

5 Example Implementation

In this section we show an example implementation for the interface classes in Ogre3D.

5.1 Intersectable

In our current setting we said that we test for intersection other than equality when assigning the pvs entries to engine entites. Hence there can be more than one matching object per PVS entry, and there is a 1:n relationship. The typical wrapper for an *Intersectable* will therefore contain an array of entities corresponding to this PVS entry. In order to use the entities of the target engine instead of Ogre3D entities, replace *Entity* with the entity representation of the target engine.

```
// a vector of engine entities
typedef vector<Entity *> EntityContainer;

class EngineIntersectable: public GtpVisibilityPreprocessor::
                                IntersectableWrapper<EntityContainer *>
{
public:
   EngineIntersectable(EntityContainer *item): GtpVisibilityPreprocessor::
      IntersectableWrapper<EntityContainer *>(item)
   {}

   EngineIntersectable::~EngineIntersectable()
   {
       delete mItem;
   }

   int Type() const
   {
       return Intersectable::ENGINE_INTERSECTABLE;
   }
};
```

5.2 Bounding Box Converter

This is maybe the most tricky part of the integration. The bounding box converter is necessary because we have to associate the objects of the visibility solution with the objects from the engine without having unique ids. This is the interface of the *BoundingBoxConverter* .

```
/** This class assigns unique indices to objects by
    comparing bounding boxes.
*/
class BoundingBoxConverter
{
public:
/** Takes a vector of indexed bounding boxes and
    identify objects with a similar bounding box
      It will then assign the bounding box id to the objects.
      The objects are returned in the object container.

      @returns true if conversion was successful
  */
  virtual bool IdentifyObjects(
                const IndexedBoundingBoxContainer &iboxes,
                ObjectContainer &objects) const
  {
    // default: do nothing as we assume that a unique id is
    // already assigned to the objects.
    return true;
  }
};
```

We give an example of implementation of a Bounding Box Converter for Ogre3D rendering engine. It is templated in order to works with any Ogre SceneManager. Again, the implementation of this interface must be adapted for the requirements of the particular engine.

```
/** This class converts preprocessor entites to Ogre3D entities
*/
template<typename T> PlatFormBoundingBoxConverter:
    public GtpVisibilityPreprocessor::BoundingBoxConverter
{
public:
  /** This constructor takes a scene manager template as parameter.
  */
  PlatFormBoundingBoxConverter(T *sm);

  bool IdentifyObjects(const GtpVisibilityPreprocessor::
         IndexedBoundingBoxContainer &iboxes,
         GtpVisibilityPreprocessor::ObjectContainer &objects) const;

protected:

  /** find objects which are intersected by this box
  */
  void FindIntersectingObjects(const AxisAlignedBox &box,
                               vector<Entity *> &objects) const;

  T *mSceneMgr;
```

```
};

typedef PlatFormBoundingBoxConverter<OctreeSceneManager>
          OctreeBoundingBoxConverter;
```

This class is inherited from *BoundingBoxConverter*. *BoundingBoxConverters* has only one virtual function *IdentifyObjects* that must be implemented. Additionally we use a helper function *FindIntersectingObjects* that is responsible for locating the corresponding objects in the scene. Let's now have a look at the implementation of *IdentifyObjects* for Ogre3D.

```
template<typename T>
bool PlatFormBoundingBoxConverter<T>::IdentifyObjects(
       const GtpVisibilityPreprocessor::IndexedBoundingBoxContainer &iboxes,
       GtpVisibilityPreprocessor::ObjectContainer &objects) const
{
  GtpVisibilityPreprocessor::IndexedBoundingBoxContainer::
  const_iterator iit, iit_end = iboxes.end();

  for (iit = iboxes.begin(); iit != iit_end; ++ iit)
  {
    const AxisAlignedBox box =
      OgreTypeConverter::ConvertToOgre((*iit).second);

    EntityContainer *entryObjects = new EntityContainer();

    // find all objects that intersect the bounding box
    FindIntersectingObjects(box, *entryObjects);

    EngineIntersectable *entry = new EngineIntersectable(entryObjects);
    entry->SetId((*iit).first);

    objects.push_back(entry);
  }
  return true;
}
```

The function just loops over the bounding boxes of the PVS entries and finds the entities that are intersected by the bouding boxes. Let's have a look now at the function *FindIntersectingObjects*, which is searching the intersections for each individual box.

```
template<typename T>
void PlatFormBoundingBoxConverter<T>::FindIntersectingObjects(
       const AxisAlignedBox &box,
       EntityContainer &objects) const
{
  list<SceneNode *> sceneNodeList;

  // find intersecting scene nodes to get candidates for intersection
  // note: this function has to be provided by scene manager
  mSceneMgr->findNodesIn(box, sceneNodeList, NULL);

  // convert the bounding box to preprocessor format
  GtpVisibilityPreprocessor::AxisAlignedBox3 nodeBox =
    OgreTypeConverter::ConvertFromOgre(box);
```

```
  // loop through the intersecting scene nodes
  for (sit = sceneNodeList.begin(); sit != sceneNodeList.end(); ++ sit)
  {
    SceneNode *sn = *sit;
    SceneNode::ObjectIterator oit = sn->getAttachedObjectIterator();

    // find the objects that intersect the box
    while (oit.hasMoreElements())
    {
      MovableObject *mo = oit.getNext();

      // we are only interested in scene entities
      if (mo->getMovableType() != "Entity")
        continue;

      // get the bounding box of the objects
      AxisAlignedBox bbox = mo->getWorldBoundingBox();

      // test for intersection (note: function provided of preprocessor)
      if (Overlap(nodeBox, OgreTypeConverter::ConvertFromOgre(bbox)))
      {
        objects.push_back(static_cast<Entity *>(mo));
      }
    }
  }
}
```

Note that the implementation of this function is maybe the one that differs the most for another engine, as it is highly depending on the particular engine design. For the Ogre3D implementation, we use a two stage approach. First we find the intersecting scene nodes. We apply a search function that is optimized for this engine. In case of Ogre3D, this is the function *findNodesIn*. The engine is responsible to provide a function for fast geometric search in the scene, in order to quickly find the objects intersecting the bounding box of a PVS entry. A spatial data structure like Octree or Kd tree is very useful in this regard. Second we traverse through the list of entities attached to the scene node. The intersection test is then applied for each indiviual bounding box.

6 View Cells Usage

By now the view cells should be accessible within the target engine. The view cells manager provides the necessary functionality to handle the view cells. In order to query the current view cell, use the following function of the view cells manager.

```
ViewCell *GetViewCell(const Vector3 &point, const bool active = false) const;
```

In your engine, the function will called like this.

```
ViewCell *currentViewCell = viewCellsManager->GetViewCell(viewPoint);
```

viewPoint contains the current location of the player. It must be of type *GtpVisibilityPreprocessor::Vector3*. In order to traverse the PVS of this view cell, we apply a PVS iterator, like in the following example. For the implementation in another engine, *Entity* from Ogre3D must be replaced by the target engine entities.

```
GtpVisibilityPreprocessor::ObjectPvsIterator pit =
  currentViewCell->GetPvs().GetIterator();
```

```
while (pit.HasMoreEntries())
{
  GtpVisibilityPreprocessor::ObjectPvsEntry entry = pit.Next();
  GtpVisibilityPreprocessor::Intersectable *obj = entry.mObject;

  EngineIntersectable *oi = static_cast<EngineIntersectable *>(obj);
  EntityContainer *entries = oi->GetItem();

  EntityContainer::const_iterator eit, eit_end = entries->end();

  for (eit = entries->begin(); eit != eit_end; ++ eit)
  {
    Entity *ent = *eit;

    // do something, e.g., set objects visible
  }
}
```