ADVANCED TOOLS FOR DEVELOPING HIGHLY REALISTIC COMPUTER GAMES

FINISHED VISIBILITY MODULES

Document identifier:	**GameTools-3-D3.4-03-1-1-Finished Visibility Modules**
Date:	**28/03/2007**
Work package:	**WP03: Visibility**
Partner(s):	**VUT**
Leading Partner:	**VUT**
Document status:	**Approved**
Deliverable identifier:	**D3.3**

Abstract: This technical report describes the final working modules of the visibility work package.

Delivery Slip

	Name	Partner	Date	Signature
From	Michael Wimmer	VUT	10.3.2007	
Reviewed by	Moderator and reviewers	ALL		
Approved by	Moderator and reviewers	ALL		

Document Log

Issue	Date	Comment	Author
1-0	02-03-2007	First draft	Micheal Wimmer
1-1	08-03-2007	Final Version	Micheal Wimmer

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files / URL
Word	

CONTENT

1. INTRODUCTION

1.1. OBJECTIVES OF THIS DOCUMENT

This document describes the finished modules for the Visibility Work Package. Its aim is to describe the modules and provide a manual for their usage. In the appendix we explain the algorithmic details of the modules.

1.2. DOCUMENT AMENDMENT PROCEDURE

Any project partner may request amendments but each amendment must be analysed and approved by the GameTools Project Coordinator or Project Manager.

1.3. TERMINOLOGY

Glossary

GTP	GameTools Project
PC	Project Coordinator
PM	Project Manager
WP	Work Package

2. USAGE MANUAL FOR THE VISIBILITY MODULES

The visibility work package (WP3) consists the following modules:

- View space partitioning (task 3.1)
- Computation of Potentially Visible Sets (PVS computation - task 3.2)
- Online visibility culling (task 3.3)

This section describes the setup and usage of the developed modules. Firstly, the view space partitioning module and the PVS computation modules are discussed in Section 2.1. As both these modules address preprocessed visibility we discuss the two modules together. Section 2.2 then discusses the usage of the online visibility culling module.

2.1. USAGE MANUAL FOR PREPROCESSED VISIBILITY

Preprocessed visibility, including view space partitioning and PVS computation, is implemented as a standalone application. The view space partitioning takes as input a static part of the scene and outputs a set of view cells. The PVS computation uses these view cells to determine potentially visible sets which are saved in a single XML file together with the view cells. Additionally there is a library version of the view space partitioning module, which can be used as a runtime interface to the preprocessed data. By linking with the view space partitioning library this module also allows the engine to use the preprocessed data: it can load the XML view cells and the PVS description and use these in runtime to further speedup rendering or schedule prefetching for out-of-core or network-based rendering.

2.1.1. Introduction

This manual provides a quick guide to generate a visibility solution, and apply the preprocessed visibility in a particular target engine. It will start with instructions 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.

2.1.2. Build Project

Open *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*.

2.1.3. Generate Visibility Solution

All necessary scripts (unix shell scripts) are located in the *Preprocessor/scripts* directory. The easiest way to use the scripts is by the way of the Unix emulation package 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 are used for view cell construction. Fewer 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.

2.1.4. Integration of Preprocessed Visibility

Requirements

Add the following libraries to your application.

- Preprocessor.lib
- zdll.lib
- zziplib.lib
- xerces-c_2.lib
- devil.lib
- glut32.lib
- OpenGL32.Lib
- glu32.lib

- glew32.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

In order to employ the preprocessor in a target engine we must make the visibility solution (PVS data) available to the engine. For this purpose we associate the entities of the engine with the PVS entries from the visibility solution. For this purpose 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 three constructs that need explanation, the *BoundingBoxConverter* and the *ObjectContainer*, and the *ViewCellsManager*.

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 and avoids the nightmare of keeping track of volatile object ids.

ObjectContainer

The object container is 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 a class derived from Intersectable which wraps one or more entities of the particular engine (e.g., through a list of pointers).

ViewCellsManager

The loading function returns a view cells manager.

```
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 accessible in the engine.

2.1.5. Example Implementation

Example Implementation

In the following paragraphs we show an example implementation for the interface classes in Ogre3D. First for the object class of the preprocessor, *Intersectable*, then for the converter class *BoundingBoxConverter*.

Class 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 object of type *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;
   }
};
```

Class BoundingBoxConverter

This is the most involved part of the integration, but it shouldn't be too difficult, either. A 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 class *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 *BoundingBoxConverter* for the Ogre3D rendering engine. It is templated in order to work with any *SceneManager* in Ogre3D (if you are not familiar with the *SceneManager* concept in Ogre3D, it does not matter). Again, the implementation of this interface must be adapted for the requirements of the particular engine.

```
/** This class converts preprocessor entities 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*. This class 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 bounding boxes. Additionally we introduce a new class, the *OgreTypeConverter*. This class is just responsible for converting basic classes like *Vector3* or *AxisAlignedBox*. between Ogre3D types and Preprocessor types. Now we revisit the function *FindIntersectingObjects*, which searches 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 other engines, 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 for providing 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 the list of entities attached to the scene node. The intersection test is then applied for each individual bounding box.

2.1.6. View Cells Usage

At this point 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 the target engine, the function is invoked like this.

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

viewPoint contains the current location of the player in the scene. 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
  }
}
```

Figure 1, The PVS Computation Module integrated in the Ogre3d engine. A snapshot of Vienna is shown, with the rendered PVS shown in the lower right corner. Note how the PVS closely fits to the geometry actually seen.

2.1.7. Setting up the environment for preprocessed visibility

This section gives an overview of how to set the most important parameters to specify the pre-processor environment. For a quick start, the short manual is sufficient. In this section we show how some of the parameters can be tweaked to provide some means for optimization.

The modules use a number of parameters which can be specified by the user through the "environment file" or command line parameters. The parameters can have one of the following types: string, int, float, and boolean. All parameters have implicit values set by the preprocessor (see the Environment.cpp source file). This value can be redefined by the environment file (see for example preprocess_visibility.env) and it can be further overwritten by a command line argument. There are two possibilities how to specify the command parameter: using a shortcut or define construct. The define construct uses the following syntax: -Dparameter_name=value. With the exception of Boolean parameters the shortcut version is specified as follows: -parameter_shortcut=value. The boolean parameter shortcut is directly followed by + or - sign (e.g. -preprocessor_use_gl_render+). The usage of the parameters is illustrated in the example scripts and environment files (preprocess_visibility, preprocess_visibility.env, generate_viewcells, generate_viewcells.env). An example of a part of the environment file is given in Figure 2.

```
Scene {
#filename ../data/vienna/vienna-buildings.x3d
```

```
#filename ../data/soda/soda.dat
filename ../data/vienna/vienna_cropped.obj
#filename ../data/PowerPlant/ppsection1/part_a/g0.ply
}
Preprocessor {
    # stored sample rays
    samplesFilename rays.out
    useGlRenderer false
    useGlDebugger false
    type rss
    detectEmptyViewSpace false
    quitOnFinish true
    computeVisibility true
    applyVisibilityFilter false
    applyVisibilitySpatialFilter true
    visibilityFilterWidth 0.01
    visibilityFile visibility.xml
}
```

Figure 2. Example of a part of the environment file. Note that the parameters can be specified using a hierarchical syntax with nested braces. Lines staring with '#' are treated as comments.

Below we give a list of the most important parameters. Each parameter in the list contains a name of the parameter (used in environment file), followed by its command line shortcut (in round brackets), and a parameter type (in square brackets). For more parameters see the environment file (generate_viewcells.env) and the documentation included in the source code of the module. In the following we describe some important options, first the parameters which are common for all modules than the options specific for each module.

2.1.7.1. Common parameters for all techniques

The following parameters are common for all modules, e.g., pvs computation module, and view space as well as view space / object space partition module.

Scene.filename (-scene_filename=) [string]

Scene description file. Currently simplified obj (only triangles), simplified X3D (.x3d), Unigraphics (.dat), UNC (.ply) formats are supported. This option has to be specified!

Preprocessor.type (-preprocessor=) [string, one of: combined, vss, rss, gvs, sampling, render]

Type of the preprocessor to use. Currently only the “combined" type is supported for PVS computation. For view cells and view / object space partition the "vss" preprocessor should be used (vss stands for View Space Subdivision).

Preprocessor.detectEmtpyViewSpace (-preprocessor_use_gl_renderer) [boolean]

Empty view space detection allows more efficient sampling for scenes with properly modelled watertight objects (those which can be correctly rendered with backface culling on). This option is beneficial for all modules and should only be omitted for scenes where there is ill-defined geometry.

ViewCells.type (-view_cells_type=) [string, one of: vspBspTree, ...]

Type of view cells partition method. Use "vspBspTree", which corresponds to our optimized view cell partition, and “vspOspTree” for the combined view space / object space partition.

ViewCells.filename (-view_cells_filename=) [string]

Specifies the filename of the view cells.

ViewCells.exportToFil (-view_cells_export_to_file=)[boolean]

Must be true for the view space (view / object space) partition methods.

ViewCells.loadFromFile (-view_cells_load_from_file=) [boolean]

Must be true for preprocessing visibility, as our visibility solver needs a set of view cells to work.

KdTree.Termination.maxNodes (-kd_term_max_nodes=) [int]

Maximum number of KD-tree nodes. The KD tree is constructed at the start of the pre-processor and it is used for ray casting acceleration and optionally for PVS storage.

KdTree.Termination.minCost (-kd_term_min_cost=)

Minimal number of triangles per node of the KD-tree so that the node is further subdivided.

2.1.7.2. Parameters for the View Space Partition module

The view space partitioning is implemented as a standalone application. Alternatively the module can be compiled as a library which can be linked to a 3rd party code. An example of such an application is the online occlusion culling module (integrated with OGRE engine) which links the library in order to load the preprocessed data and use them in runtime. This section gives an overview of the important parameters of the view space partitioning module

ViewCells.type (-view_cells_type=) [string]

Use "**vspBspTree**", which corresponds to our optimized view cell partition

VspBsp.Termination.maxViewCells (-vsp_bsp_term_max_view_cells=) [int]

Maximal number of generated view cells by the subdivision

VspBsp.Construction.samples (-vsp_bsp_construction_samples=) [int]

Number of visibility samples to be used for the construction of the view cell subdivision.

VspBsp.Termination.minGlobalCostRatio (-vsp_bsp_term_min_global_cost_ratio=) [float]

Defines the minimal render cost decrease where the view cell subdivision is terminated

VspBsp.maxPolyCandidates (-vsp_bsp_max_poly_candidates=) [int]

Number of geometry aligned splits evaluated for the next best split.

VspBsp.useCostHeuristics (-vsp_bsp_use_cost_heuristics=) [bool]

If spatial mid split is always taken for axis aligned splits or split plane is evaluated using the cost model.

ViewCells.PostProcess.merge (-view_cells_post_process_merge=) [bool]

If the final subdivision should be merged according to the cost model to optimize the view cells.

ViewCells.Construction.samples (-view_cells_construction_samples=) [int]

The number of samples cast before the final merge step (the subdivision is handled by VspBsp.Construction.samples)

2.1.7.3. Parameters for the combined View / Object Space Partition

We have implemented an optimized version of the view cell partition module that interleaves object space and view space partitioning and thus generates even better view cells. In the following we describe the parameters for handling this version.

ViewCells.type (-view_cells_type=) [string]

Use "**vspOspTree**", which corresponds to our optimized view space / object space partition

We describe the global parameters influencing the construction and interleaving of both hierarchies (view and object space).

Hierarchy.Construction.type (-hierarchy_construction_type=) [int]

0 = sequential computation (fast), 2 = optimized interleaved method

The sequential computation subdivides view space first using surface area heuristics, then the view space using our optimized view space partition. The interleaved method computes view and object space subdivision in an interleaved fashion, aiming for the optimal #leaves in both domains.

Hierarchy.Construction.samples (-hierarchy_construction_samples =) [int] [e.g., 1000000]

#samples used for construction of hierarchy

Hierarchy.Construction.maxRepairs(-hierarchy_construction_max_repairs =) [int] [e.g., 1000]

This option highly influences computation time for the interleaved method. Maximum #repair steps per subdivision step in one domain. A lower value means faster but less accurate computation.

Hierarchy.Termination.maxMemory (-hierarchy_construction_max_memory =) [int] [e.g., 20)]

Maximal memory cost of visibility solution in MB. Note that this value will be higher for the final visibility solution as we are only used coarse sampling here.

Hierarchy.Termination.maxLeaves (-hierarchy_termination_max_leaves =)[int] [e.g., 3000000]

Maximal number of leaves for both subdivisions (e.g., sum of view cells and object space hierarchy leaves)

The following options concern the local and global parameters concerning only the **view space partition** (**vsp**).

VspTree.Termination.maxViewCells (-vsp_term_max_viewcells =) [int] [e.g., 5000]

The maximal number of leaves in the view space partition.

VspTree.useCostHeuristics (-vsp_use_cost_heuristics =) [boolean]

If our optimized view space partition algorithm should be used.

VspTree.splitUseOnlyDrivingAxis (-vsp_split_only_driving_axis =) [boolean]

If only the driving axis (i.e., the longest axis in the current parent cell) should be subdivided.

The following options concern the local and global parameters concerning only the **object space partition.** We use a bounding volume hierarchy as object space hierarchy.

BvHierarchy.Termination.maxLeaves (-bvh_term_max_leaves =) [int] [e.g., 20000]

The maximal number of leaves in the object space partition.

BvHierarchy.useCostHeuristics (-bvh_use_cost_heuristics =) [boolean]

If some sort of cost heuristics (**surface area**, **sampled visibility based heuristics**) should be used.

BvHierarchy.useSah (-vsp_use_sah =) [boolean]

If **surface area heuristics** should be used or **sampled visibility based heuristics**.

BvHierarchy.splitUseOnlyDrivingAxis (-bvh_split_only_driving_axis =) [boolean]

If only the driving axis (i.e., the longest axis in the current parent cell) should be subdivided.

2.1.7.4. Parameters for the PVS Computation Module

The PVS computation module is implemented as a standalone application. The parameters for the module can be specified by the user through the "environment file" or on the command line as described for the view space partitioning module. Below we give a list of the most important parameters. For other parameters see the environment file (*preprocess_visibility.env*) and the documentation included in the source code of the module.

Preprocessor.type (-preprocessor=) [string]

The type of the preprocessor to use. Can be one of: combined, rss, sampling, gvs. Suggested value: combined.

Combined provides a mix of a sampling-based method (PGV, Section 3.3) and an approach using a simple ray space subdivision. Instead of an exact solution, we also provide guided visibility sampling (see Section 3.4), which is not a progressive method, but aims for a very accurate visibility solution, comparable to an exact algorithm.

Preprocessor.totalSamples (-total_samples=) [int]

The total number of samples cast.

Preprocessor.samplesPerPass (-samples_per_pass=) [int]

Number of samples to cast per pass.

RssPreprocessor.distributions (-rss_distributions=) [string]

Distributions to use for mixture distribution sampling separated by + sign. Can consist of rss, object, spatial, global, direction, object_direction, reverse_object, reverse_viewspace_border, mutation. Suggested value: -rss_distributions=object_direction+spatial+mutation

Preprocessor.detectEmptyViewSpace (-preprocessor_detect_empty_viewspace) [bool]

Turn on/off the detection of empty space (see the description of the PVS computation algorithm). The empty view space detection can be very beneficial, but it assumes a scene with correct orientation of normals with watertight objects.

ViewCells.useKdPvs (-view_cells_use_kd_pvs) [bool]

Store PVS as entries of a KD-tree (on) or BVH (off). Storing PVSs as KD-tree entries usually converges faster in terms of accuracy of the PVS.

Preprocessor.useGlRenderer (-preprocessor_use_gl_renderer) [boolean]

Tells the preprocessor to open an OpenGL window which serves for visualization, testing and visual debugging of the preprocessor. This functionality is currently implemented using Qt OpenGL widget and thus requires compiling the preprocessor with Qt library.

Preprocessor.applyVisibilityFilter (-preprocessor_apply_visibility_filter) [boolean]

Tells the preprocessor if the visibility filter should be applied.

Preprocessor.visibilityFilterWidth (-preprocessor_visibility_filter_width=) [float]

Width of the visibility filter specified as a ratio of the size of the bounding box of the scene. A reasonable value for common scenes is 0.01.

Preprocessor.visibilityFile (-preprocessor_visibility_file=) [string]

Name of the file into which the preprocessed visibility information should be exported.

For the guided visibility sampling (=gvs) preprocessor, we have to set different parameters. Note that the algorithm runs on triangle level, but the triangle pvs is converted to the given granularity of objects (or kd cells, respectively, if the "view_cells_use_kd_pvs" parameter is set) each time after we finished processing a view cell.

GvsPreprocessor.gvsSamplesPerPass (-gvs_samples_per_pass=)[int]

Samples shot in one pass of the algorithm. This option sets the termination criterium in combination with the parameter value for the minimal contribution.

GvsPreprocessor.gvsMinContribution (-gvs_min_contribution=)[int]

For each view cell, the gvs algorithm terminates if after a number of rays given by "gvs_samples_per_pass", the number of newly found triangles is less than "gvs_min_contribution".

2.2. MANUAL FOR THE ONLINE VISIBILITY CULLING MODULE

The online visibility culling module is implemented as a specialized scene manager. It has been integrated into the Ogre3D and Shark3D engines.

2.2.1. Implementation

The major focus in the implementation of the online culling module was a careful separation of the CHC algorithm used to do visibility culling from the platform dependent code, which allows reusing the same algorithm code for both Ogre3D and Shark3d game engines.

The main class of the module is the *CullingManager*. The particular algorithm is realized as class which inherits from *CullingManager* and implements the RenderScene method. The CHC algorithm is implemented by the *CoherentCullingManager* class. The class diagram of the integration into Ogre3D is given in Figure 3.

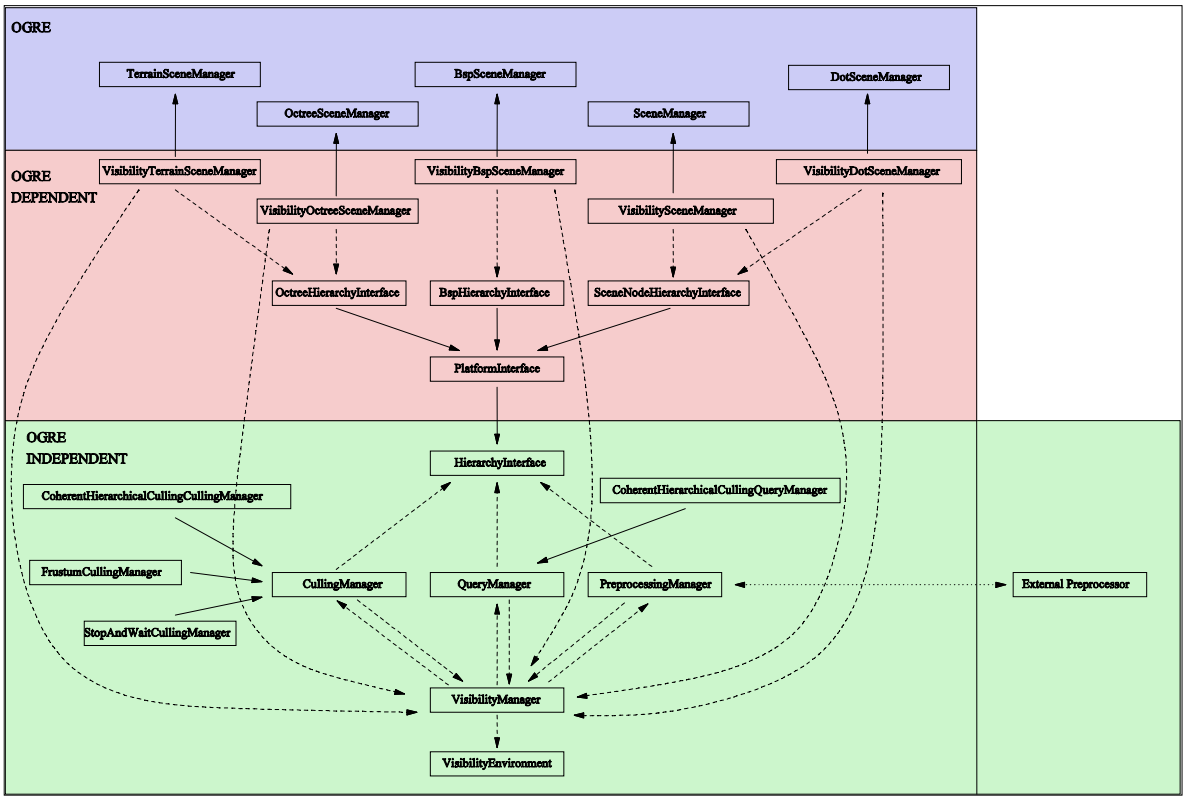

Figure 3. Class diagram depicting the integration of the online culling module into Ogre engine.

The Ogre3D implementation was realized using a specialized scene manager plugin. We implemented the single pass version of the algorithm as well as a version which applies an initial depth pass. The algorithm runs nearly orthogonal to other Ogre3D features and supports Ogre texture-based shadows as well as shadow volumes. The second functionality of the online module is the integration of the preprocessed visibility: it provides loading and using the preprocessed view cells in Ogre. The view cells are loaded from an XML file which is computed by the PVS computation module. The scene objects corresponding to the PVS for the current view point are set to visible while all other objects are set invisible and thus not rendered. The pseudocode of the CHC algorithm is given in Figure 4.

```
TraversalStack.Push(hierarchy.Root);
while ( not TraversalStack.Empty() or
        not QueryQueue.Empty() )
{
```

```
  //-- PART 1: process finished occlusion queries
  while ( not QueryQueue.Empty() and
         (ResultAvailable(QueryQueue.Front()) or
          TraversalStack.Empty()) )
  {
    node = QueryQueue.Dequeue();
    // wait if result not available
    visiblePixels = GetOcclusionQueryResult(node);
    if ( visiblePixels > VisibilityThreshold )
    {
      PullUpVisibility(node);
      TraverseNode(node);
    }
  }

  //-- PART 2: hierarchical traversal
  if ( not TraversalStack.Empty() )
  {
    node = TraversalStack.Pop();
    if ( InsideViewFrustum(node) )
    {
      // identify previously visible nodes
      wasVisible = node.visible and
                   (node.lastVisited == frameID - 1);
      // identify nodes that we cannot skip queries for
      leafOrWasInvisible = not wasVisible or IsLeaf(node);
      // reset node's visibility classification
      node.visible = false;
      // update node's visited flag
      node.lastVisited = frameID;
      // skip testing previously visible interior nodes
      if ( leafOrWasInvisible )
      {
        IssueOcclusionQuery(node);
        QueryQueue.Enqueue(node);
      }
      // always traverse a node if it was visible
      if ( wasVisible )
        TraverseNode(node);
    }
  }
}

TraverseNode(node)
{
  if ( IsLeaf(node) )
    Render(node);
  else
    TraversalStack.PushChildren(node);
}

PullUpVisibility(node)
{
  while (!node.visible)
  {
```

```
    node.visible = true;
    node = node.parent;
  }
}
```

Figure 4. **CHC algorithm pseudocode.**

2.2.2. CHC integration manual

In the following we decribe the steps necessaray to implement CHC into a particular target engine. In principle the user needs to implement the methods of a hierarchy interface class to work on the target rendering engine.

Requirements

Add this library to your application.

- GtpVisibility.lib

The library can be found in the following directory.

- trunk/Lib/Vis/Preprocessing/lib/$(ConfigurationName)

The class contains the interface class *HierarchyInterface* This is an abstract class that enables us to keep the occlusion culling algorithms independent from the platform, The user has to provide an implementation of the interface methods for a particular target engine. The HierarchyInterface stores the current frame number as an internal state.

Most of the hierarchy interface methods operate on a *HierarchyNode* *. This is merely a void * in order to achieve maximum flexibily. The user has to use the node class of the target engine as argument whenever a *HierarchyNode* * is required.

Each *HierarchyNode* has to provide a flag isVisible as well as an integer timestamp *lastVisited.*

The methods *HierarchyInterface* class that must be reimplemed for using CHC are:

OcclusionQuery* IssueNodeOcclusionQuery (HierarchyNode * *node*, *const bool wasVisible*)

```
Issues an occlusion query for this node and returns the query. wasVisible
indicates if the node was visible in the last frame – in this case, the
geometry may be tested instead of the geometry at the discretion of the
implementation.
```

*void TraverseNode(HierarchyNode * node)*

Renders and traverses the given hierarchy node. Traversal should be done by pushing the child nodes onto a DistanceQueue which can be referenced via GetQueue().

void RenderNode(HierarchyNode * *node*)

Renders the given hierarchy node. The render call should make sure that geometry is only rendered once in a frame, even if this function is called more than once.

*void PullUpVisibility (HierarchyNode * node) const*

Pulls the visibility classification up the hierarchy, i.e., beginning from this node, all parents are set to visible because a child node is visible.

*void SetNodeVisible (HierarchyNode * node, const bool visible) const*

Sets the node to be visible / invisible in the current frame.

*void SetLastVisited (HierarchyNode * node, const unsigned int frameId)*

frameid specifies the frame number at which the node was visited the last time and should be stored in the HierarchyNode.

*bool IsLeaf(HierarchyNode * node) const*

Returns true if the node is a leaf, false if the node is an interior node of the hierarchy.

*float GetSquaredDistance(HierarchyNode *node) const*
Returns distance of the node to the view plane.

*bool CheckFrustumVisible(HierarchyNode *node, bool &intersects)*
Checks if the node is visible from the current view frustum.
The parameter intersects returns true if the current node intersects the near plane. This is necessary to skip queries in case the nodes intersect the near plane.

*bool HasGeometry(HierarchyNode *node) const*
Returns true if there is renderable geometry attached to this node

The hierarchy interface provides some more important functions for hierarchy processing:

DistanceQueue GetQueue ( )*

Returns pointer to the priority queue storing the nodes based on the distance to the view point in a front-to-back order.

*void SetHierarchyRoot(HierarchyNode *root)*
sets the root of the scene hierarchy. Needs to be called once.

As an example we show the implementation of the TraverseNode method for an Octree hierarchy traversal (using the OctreeSceneManager), This method does a traversal from a particular octree node to the leaves of the hierarchy. In each step the renderable geometry is rendered. The user must proceed in a similar fashion to implement all the functionality of the HierarchyManager for his engine.

```
void OctreeHierarchyInterface::TraverseNode(GtpVisibility::HierarchyNode
*node)
{
	++ mNumTraversedNodes; // just for stats

	// A cast from hierarchy node to the node class of the engine
	Octree *octree = static_cast<Octree *>(node);

	// Render the node.
	RenderNode(node);

	// Continue traversal for all children of the octree.
	if (!IsLeaf(node))
	{
		Octree *nextChild;

		if (!IsLeaf(node))
		{
			Octree *nextChild;

			for (int z = 0; z < 2; ++ z)
			{
				for (int y = 0; y < 2; ++ y)
				{
					for (int x = 0; x < 2; ++ x)
					{
						nextChild = octree->mChildren[x][y][z];

						// push on queue for further traversal
						if (nextChild)
							GetQueue()->push(nextChild);
					}
				}
			}
		}
	}
}
```

2.2.3. OcclusionCullingSceneManager

The *OcclusionCullingSceneManage*r provides several options for online and offline occlusion culling. These options must be provided in the form of an Ogre configuration file. The configuration file must be passed as parameter of the scene manager function *setWorldGeometry***.** We provide an example configuration file *"terrainCulling.cfg*" in *Lib\Vis\OnlineCullingCHC\scripts*.

```
##  Online culling algorithm settings
OnlineCullingAlgorithm=CHC
#OnlineCullingAlgorithm=SWC
#OnlineCullingAlgorithm=VFC
#OnlineCullingAlgorithm=DEFAULT

##  Scene geometry: can be one of *.iv, *.obj
## terrain: generate terrain
#Scene=terrain
Scene=../../../../../resources/media/city1.iv;../../../../../resources/medi
a/roofs_1500.iv;../../../../../resources/media/CityRoads60.iv;../../../../.
./resources/media/CityPlane60.iv

## visibility solution (does not work with terrain yet)
ViewCells=../../../../../../../GTP/trunk/Lib/Vis/Preprocessing/scripts/vien
na-visibility.xml.gz


## depth pass first renders object without material to fill depth buffer
UseDepthPass=no

## performance option: the render queue will be flushed after some frames
FlushQueue=yes


## Settings for terrain scene
## these settings are inherited from terrain scenemanager

…
```

Figure 5, Example for a configuration file for the OcclusionCullingSceneManager

In Figure 5 we can see such an example configuration. The "*Scene*" parameter provides either static geometry or a terrain generated from a heightmap. Note that the visibility solution can be applied NOT only on the static geometry provided with the "Scene" parameter, but to any entities in the engine. In addition to the startup configuration, some parameters can be set interactively in the scenemanager using the scenemanager function

```
setOption(const String & key, const void * val)
```

All options take either bool or int as second argument. The most important parameters are:

Algorithm [int]
Sets the applied visibility algorithm.

UseViewCells [boolean]
If the previously loaded view cells should be used for preprocessed per view cell visibility culling. off per default.

UseDepthPass [boolean]
If depth only pass should be applied first.

ExecuteVertexProgramForAllPasses [boolean]
Together with depth pass, if there are problems with vertex shaders that change visibility. renders vertex shaders also in depth pass.

Threshold [int]
The threshold of visibility pixels where an object is still considered invisible. Default is 0. Warning: a value different from zero means that there will be visibility errors.

AssumedVisibility [int]
The #frames where visibility is assumed to be coherent. Default is 1 frame. This value influences performance. For slowly changing scenes this value can be set higher.

TestGeometryForVisibleLeaves [boolean]
This value tests the geometry instead of the bounding boxes for leaf nodes. This option improves performance, but cannot be used together with transparent objects.

FlushQueue [boolean]
If the render queue should be flushed after some frames (often beneficial for performance).

3. APPENDIX – ALGORITHM DESCRIPTIONS

3.1. VIEW SPACE PARTITIONING

3.1.1. Overview

The view space partitioning module deals with subdividing the space of all possible viewpoints and viewing directions into view cells. The result of the computation is a BSP tree describing the geometry of the view cells and the view cell hierarchy describing the view cells at different levels of detail. Note that on the contrary to common approaches the BSP tree is constructed according to the actual visibility in the scene: it aims to minimize the average rendering time for the PVS associated resulting set of view cells. The process of view space partitioning consists of three main steps:

- Visibility sampling
- View space subdivision
- View space merging

The first step estimates visibility in the scene, which allows basing the view cell construction on scene visibility without incurring the overhead of having to calculate complete visibility. We use stochastic sampling by casting rays that are uniformly distributed in the whole view space. As a result we obtain a set of maximal free line segments which we call *visibility segments*. The visibility segments provide information about scene visibility for the subsequent steps of the view cells construction.

The second step performs an adaptive hierarchical subdivision of view space. The subdivision is driven by heuristics which aim to minimize the estimated rendering cost of the resulting subdivision. The result is set of elementary view cells which satisfy certain termination criteria. We choose a very fine subdivision in order to allow more choices for the subsequent merging step.

The third step merges the elementary view cells to larger ones while minimizing the increase of the estimated rendering cost for each merging step. The merging progress is recorded in a *merge history tree*. The merge history defines a *view cell hierarchy*, allows retrieving an optimal set of view cells for a specified granularity of the view space subdivision. Additionally, this hierarchy can be used to compress the PVSs in a simple and efficient way. The merging step will implicitly approximate important visibility events in the scene. The three steps of our algorithm are illustrated in **Figure 6**.

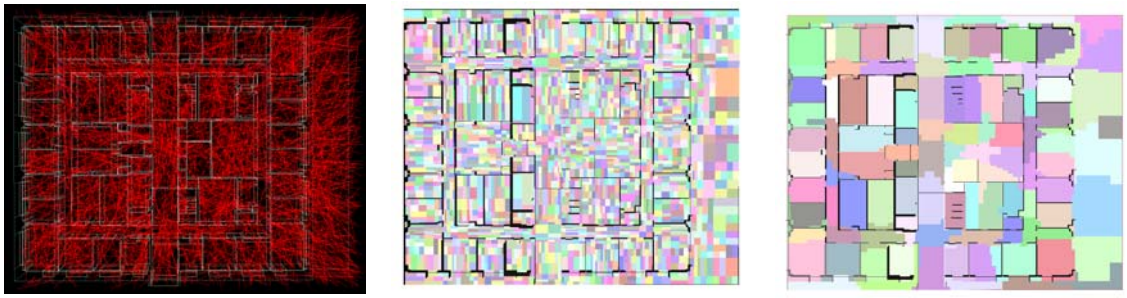

Figure 6, Illustration of the view space partitioning process in a building interior (soda scene). (left) Visibility sampling. (center) Subdivision into elementary view cells. (right) Merged view cells

3.1.2. Cost Model

$$c(\mathcal{S}) = \sum_{i \in \mathcal{S}} p(i) r(PVS_i),$$

where $r(PVS_i)$ is a rendering time estimation for the approximate PVS of cell i and p(i) is the probability of the viewpoint being located in view cell i. Assuming that viewpoints will be distributed uniformly in the whole view space, p(i) can be chosen as the ratio of the volume V_i of the given view cell and the total volume V of view space:

$$p(i) = \frac{V_i}{V}.$$

In general, the user can specify any probability density d for viewpoint locations, so that areas where the user is more likely to move receive more attention in the view cell construction (note that this feature is not yet supported by the module). p(i) is then given by:

$$p(i) = \frac{\int_{x \in i} d(x)}{\int d(x)}.$$

The rendering time for a view cell is estimated from the rendering times for the objects in the PVS:

$$r(PVS) = \sum_{o \in PVS} r(o).$$

The rendering time estimation function r(o) is difficult to establish exactly since it depends not only on the particular set of objects and their attributes, but also on the actual implementation and hardware. On the other hand, the view cell subdivision should not be tied too much to a specific hardware, neither do we have an accurate PVS to determine the absolute value of the rendering time. Therefore we propose to loosely calibrate an analytic rendering time estimation function to a small number of target machines. Since current graphics hardware is CPU limited for small batches, the following function provides good results:

$$r(o) = \max(a, bt_o, cp_o),$$

where a, b and c are positive constants, and t_o and p_o are the number of triangles and the number of projected pixels of object o (estimated from some points in the cell) respectively.

3.1.3. Representation of View Cells

We maintain the view space partition as a binary space partition tree which is constructed top-down. The leaves of this tree are convex polyhedra which form a set of elementary view cells. The final view cell partition is constructed from these elementary view cells using a bottom-up merging procedure which is recorded in the merge history tree. Note that merging is not tied to the original subdivision and therefore usually results in a completely different, more optimal tree. Both steps of the view cell hierarchy construction will be detailed in the next section. The representation of view cells using the two hierarchies is shown in Figure 7.

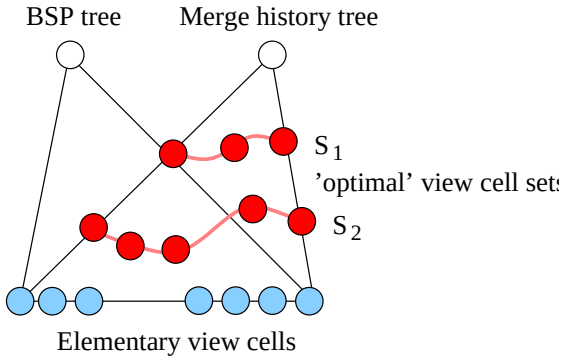

Figure 7. The view cell hierarchy is represented using a BSP tree and the merge history tree. The BSP tree provides geometrical description for elementary view cells. The merge history tree provides logical grouping of the view cells which allows extracting set of view cells with specified granularity of the subdivision. The example shows two sets S_1 and S_2 where the desired number of view cells for S_2 is larger than for S_1.

Note that we partition the view space only in the spatial domain, since the observer can quickly move through the whole directional space within a few frames by changing her viewing direction. Fortunately, culling of directional space is efficiently handled by view-frustum culling.

3.1.4. Visibility Sampling

We gain information about global visibility in the scene by sampling the whole view space. A view space sample is a 5D entity corresponding to a ray in primal space. For simplicity, let's assume that the view space is defined by a 3D spatial box of possible ray origins (*view space box*) and contains all possible ray directions.

The view space is then sampled using the following strategy:

1. Select a point p inside the view space box and a direction d using uniform distributions.
2. Cast a `forward' ray from p in direction d and a `backward' ray from p in the opposite direction -d.
3. Construct a line segment formed by the calculated termination points of the forward and backward rays. If at least one of the two rays hits an object, we call the resulting line segment a visibility segment and store it for later use.

The determination of visibility segments is illustrated in Figure 8.

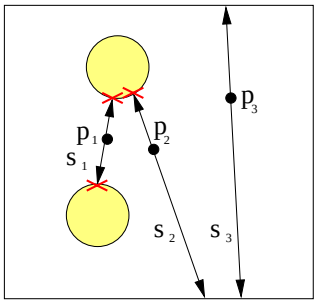

Figure 8. Illustration of determination of visibility segments. Three visibility samples are generated from points p_1, p_2 and p_3, resulting in two valid visibility segments. s_1 carries information about visibility of two objects, s_2 about one object. s_3 is not a valid visibility segment since it does not hit any object.}

There is an interesting subtlety involved in polygon orientations. For a general scene, the above procedure will create visibility segments in the interiors of objects if those regions are not explicitly exempted from view space. If, however, the input model is guaranteed to be watertight and the polygons have a consistent orientation, the algorithm can detect *empty view space* at no additional cost: if a ray hits the back side of a polygon, the ray starting point is simply shifted to the intersection point and the ray is re-cast (see Figure 9). In this way, no visibility segments will be generated in empty space. We have found that empty view space detection can improve the final view cell hierarchy, because visibility regions are more clearly separated (see the Preprocessor.detectEmptyViewSpace parameter of the module).

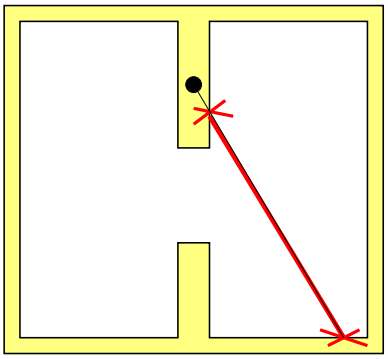

Figure 9. Example of empty view space detection. The origin of the ray lies inside the wall of the building and so the ray first hits a back-facing polygon. We shift the ray origin to the intersection point and the ray is re-cast. The resulting visibility segment is shown in red.

3.1.5. View Space Subdivision

The view space subdivision uses a top-down approach to create a set of elementary view cells. In particular we use *binary space partitioning* (BSP) maintained by a BSP tree. Starting with a single view cell corresponding to the whole view space, we recursively subdivide the current view cell using either axis-aligned planes or planes aligned with scene geometry. Each cell of the subdivision also references all visibility segments that intersect it.

The BSP construction uses a greedy optimization for the best-next split. Note that in contrast to most previous work on BSP or kD-tree construction we use global optimization procedure with a priority queue for selecting the splitting plane candidates. An entry in the priority queue consists of a reference to leaf node, the best splitting plane candidate inside this leaf, and a cost reduction which would be achieved when splitting the leaf by the plane. For each step of the subdivision we select the node for which its best splitting plane provides the highest global render cost reduction. The subdivision is thus refined progressively and regions with the highest potential render cost decrease are subdivided first.

The best splitting plane for a node is established as follows: we generate axis-aligned splitting plane candidates aligned with the endpoints of rays intersecting the node. Additionally, we generate planes aligned with the geometry contained in that node (if any). For each candidate, we calculate the reduction of the expected rendering cost c(S) that would result from subdividing the node by that plane. This is done by partitioning the current set of visibility segments according to the plane (note that a segment can be assigned to both sets), computing the PVSs (i.e., the union of the objects hit by the visibility segments) for the front and back segment sets, and using those to evaluate the reduction of the cost. Finally, we choose the candidate plane that provides the most reduction in expected rendering cost and put a corresponding entry in the priority queue.

The subdivision is terminated when one of the following termination

criteria is met:

- A specified maximum number of elementary view cells have been generated. It ensures that the algorithm stays within reasonable memory bounds.
- The cost reduction for the best splitting plane is below a specified threshold. As the cost reduction can temporarily stagnate, we only terminate when the reduction was below the threshold in several successive subdivision steps in that branch of the tree.

Due to the priority driven subdivision the view space will be evenly subdivided no matter of the termination point. This is not the case for depth-first approaches, where for example a termination on low memory would leave whole view space regions unsubdivided. Additionally, the local termination criteria used in the depth-first approach are hard to tune.

Note that the time required for the subdivision is dominated by the cost evaluation for the candidate planes. To accelerate this process, we limit the number of axis-aligned as well as geometry-aligned candidates. If the number of visibility segments or geometry planes is above these limits we select their random subsets. This speeds up the selection especially for nodes near the root of the subdivision tree.

3.1.6. View Space Merging

View space merging is a bottom-up process which aims to reduce the number of view cells while minimizing the cost of the merged view cell set. We use a greedy algorithm that always merges the pair of view cells resulting in the minimal cost increase. This is done by maintaining a priority queue of view cell merge candidates. Each pair of neighboring view cells forms a merge candidate. The relative cost increase due to the merge candidate consisting of view cells x and y is given as:

$$r(x,y) = \frac{c_r(\mathcal{S}_{merged})}{c_r(\mathcal{S})}$$

where S is the current set of view cells and S_{merged} is the set resulting from merging x and y. The priority of the merge candidate p(x,y) is given by:

$$p(x,y) = \frac{1}{r(x,y)} = \frac{c_r(\mathcal{S})}{c_r(\mathcal{S}_{merged})}$$

In the beginning, the queue is initialized with all pairs of neighboring view cells. At every step, we select the merge candidate with the highest priority (smallest relative cost increase) and merge the associated view cells. The PVS and the estimated rendering cost of the new view cell is calculated. After the merge, the priority queue is updated by removing entries corresponding to the merged view cells and inserting new entries corresponding to the created view cell and its neighbors. Note that the set of neighbors for a view cell is determined using the BSP tree.

The merging process provides a sequence of view cell sets: at every merge step we obtain a new set of view cells with exactly one view cell less than in the previous set. We record the whole merging process in a *merge history tree*. The leafs of this tree correspond to elementary view cells. Every internal node corresponds to a merged view cell. With each internal node we associate the current cost of the subdivision resulting from the corresponding merge.

Once the complete merge history tree is built, there are several ways how to create a view cell partition from the tree. The easiest way is to specify a target number n of view cells and extract these from the tree. These view cells can then be used as input for a visibility preprocessing algorithm.

3.1.7. Extracting View Cells

The view cell hierarchy allows to extract the set of view cells most suitable for the target application. Bellow we discuss three possibilities of view cell extraction.

Getting a specified number of view cells. Often, it is most convenient to specify a desired number of view cells to use for visibility calculation. This limits both the preprocessing time and the storage required for PVS data, as well as restricting the frequency with which new PVS data has to be fetched due to crossing into a new view cell at runtime. To obtain a given number of view cells, we perform a priority traversal of the merge history tree. The priority of a node is given by the cost associated with the node. When reaching a leaf node, we add it to the list of resulting view cells. When the sum of traversed leaves and the nodes in the priority queue becomes equal to the desired number of view cells, we terminate the traversal and add the contents of the priority queue to the resulting view cells. The collected view cells form a cut of the merge history tree at optimal depths with respect to the specified granularity.

Fulfilling a given memory budget. The procedure described above can be extended to allow specifying an approximate memory budget for the complete PVS representation (it is only approximate because it relies on the approximate PVS from the sampling step). At every step of the tree traversal, we can easily calculate the memory requirements for the current set of view cells and their approximate PVSs. We can terminate the traversal when the budget is reached and collect the resulting view cells as described above. Note, that this step can also be applied after computing the final visibility classification. In this case the memory budget would be the real budget for storing the PVS representation.

Extracting important view cells. An alternative to the methods described above is view cell extraction based on their estimated rendering cost. In particular we can use the maximal tolerance of the increase of the estimated rendering cost over the minimal cost, i.e. the cost provided by the densest view space partition (elementary view cells). The view cells are extracted again by a priority traversal of the merge history tree. At each step we evaluate the ratio of the current cost (stored with the processed node) and the minimal cost. If the ratio falls below a threshold, we terminate the traversal and collect the resulting view cells as described above.

Interactive specification. In practice, the user can combine these methods in an interactive setup. The selection process can easily be accomplished with a real-time visualization of the view cells and a depiction of the associated cost and memory budgets, as well as the just described cost ratio. Typically, the user would start by setting an initial cost ratio and refining the result interactively. This allows the user interactive control over the process, which is a feature often desired by practitioners.

3.2. COMBINED VIEWSPACE / OBJECT SPACE PARTITIONING

Visibility preprocessing algorithms assume that a **view space** is partitioned into a set of **view cells** and the **object space** is partitioned into a set of **objects**. In a preprocessing step, they determine for each view cell a potentially visible set of objects (PVS). In the last section we dealt only with view space partition, while in this section we present a technique that additionally deals with object space partition.

Our pre-processor handles both scenes that consist of predefined objects as well as scenes provided as triangle soup. The view space partition scheme presented in the last section is aiming scenes that are already logically partitioned into objects. The method presented in this section is beneficial in the latter case. It combines our view space partition with an equally optimized object space partition and aims to provide ratio of view space / object space splits which is optimal in terms of PVS storage cost / render time.

It is important to note that both view space and object space subdivision are visibility dependent and also interdependent. The resulting set of objects directly influences the quality of the preprocessed visibility information. If the partitioning is too fine, the memory costs for storing preprocessed visibility will be very high. Additionally the setup costs for rendering the corresponding fine-grained objects on the GPU will become a burden. On the other hand if the partitioning is too coarse, visibility information will be inaccurate, leading to more geometry being rendered than necessary, and therefore slower frame rates.

In this section we present a technique which aims to automatically generate good view space and object space subdivisions based on visibility information. As a result, the average render cost in the scene can be reduced significantly compared to a naive view space and object space subdivision. We will show that our algorithm consistently improves the render cost at a given memory cost, while naive methods are very fragile with respect to the relative depths of object and view space subdivisions. Figure 6 shows a simple illustrative example: using a non visibility-aware object subdivision method, the spherical in the room would inevitably have been needlessly subdivided uniformly, whereas our method concentrates the subdivision to the front where actual visibility events take place.

3.2.1. Algorithm outline

The main idea of the algorithm is to acquire coarse visibility information about the scene in a global sampling step, and then use this visibility information to partition view space and object space simultaneously. The proposed method thus consists of two main parts: visibility sampling and interleaved subdivision. **Visibility sampling** acquires visibility information which is represented as a

set of maximal free line segments. These segments are then used to quickly determine visibility between the cells of the constructed view space and object space partitions.

The **subdivision** starts with a single view cell representing the whole view space and a single object representing the whole scene geometry. Both the view and object space partitions are progressively refined by splitting either a view cell or an object into two parts. The main criterion driving the splits are the expected render and memory costs which are estimated using visibility samples. Each split attempts to reduce the render cost while keeping the associated memory cost increase as small as possible.

Candidates for splitting are chosen from all current view cells and object space cells. The candidates are stored in a priority queue, and at each step we pick the candidate which provides the best ratio of render cost reduction over memory increase. This candidate is then used to subdivide the associated view space or object space cell. The subdivision proceeds until the given termination criteria are reached. In particular the algorithm terminates if the local render cost reduction falls below a specified threshold, or a maximal memory budget for the whole visibility data is reached. As a result the algorithm delivers optimized view space and object space partitions which can then be fed into any from-region visibility preprocessing algorithm.

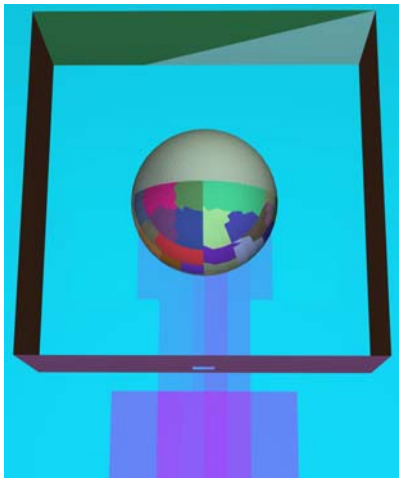

Figure 10, Example of combined view space and object space subdivisions. The scene consists of a sphere inside the room which can be seen only through a small hole. We show a cut through the 3d view space subdivision. The view space subdivision concentrates in the regions which see the sphere through the hole and therefore have higher render cost (low cost=blue, high cost=magenta). The object space subdivision focuses on the front of the sphere which is visible from more view points.

3.2.2. Evaluating the render cost reduction

The crucial part of the algorithm is to evaluate the render cost reduction dR resulting from subdividing either view space or object space.

3.2.2.1. View Space Splits

The render cost of a single view cell is defined as according to the last section:

$$c_r(V) = p(V)r(PVS_V)$$

When subdividing a view cell V by a splitting plane, the render cost reduction is given by the difference of the render cost of the new view cells and the old one:

$$dR(V) = c_r(V_b) + c_r(V_f) - c_r(V)$$

where V_b and V_f are back and front fragments of the view cell V with respect to the splitting plane.

3.2.2.2. Object Space Splits

To evaluate the render cost reduction when splitting an object, we first need to know from which view cells the object and its new fragments can be seen. We denote the set of view cells which can see an object O as PVS_O. The expected render cost of an object is then expressed as:

$$c_{r(O)} = \sum_{V \in PVS_O} \bar{r}(O,V)p(V)$$

where $p(V)$ is the probability of view point located in view cell V and $r(O, V)$ is the render cost of the object O seen from view cell . When subdividing an object O, the render cost reduction is given by:

$$dR(O) = c_r(O_b) + c_r(O_f) - c_r(O)$$

where O_b and O_f are the back and front fragments of the object. The object space split is illustrated in Figure 11.

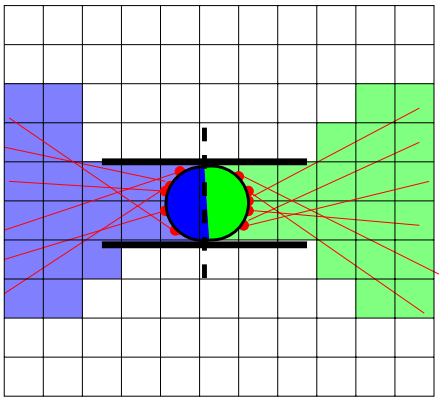

Figure 11, Illustration of an object space split A spherical object inside a tube-like structure is subdivided into two parts shown in blue and green. In this example the view cells which see the object (identified by visibility samples) break into two disjoint sets.

3.2.3. Subdivision

This section describes in detail the selection of splitting planes, determination of their processing order, and the computation of the actual splits of view space or object space nodes.

3.2.3.1. Establishing split candidates

Whenever a new (object space or view space) cell is generated in the subdivision, we establish a new splitting plane candidate for this cell and add it to the priority queue. Within the cell, we aim to find the split plane with the highest priority. Both dR and dM functions have discontinuities at places of visibility changes (changes in PVS). The visibility changes can occur only at the end-points of the visibility samples clipped to the current node, i.e., either at an end-point of a visibility sample due to an object within the view cell, or at the intersection of a visibility sample with the view cell boundary. In order to achieve a high chance of separating the PVS, we compute these points and evaluate the split priorities for the corresponding split plane positions in all three axes. The splitting plane candidate for this node is then established at the position with highest priority.

3.2.3.2. Processing split candidates

The split candidates established for every leaf of the current subdivision have to be processed in the order of descending priority. The basic algorithm described so far would just use a single priority

queue of the candidates, pick the candidate with best priority, split the corresponding node and put split candidates for the newly created nodes into the priority queue. However, a view space split

induces a change of the priorities of object space split candidates and vice versa. The affected candidates are those which could see the node that was split, i.e., those which are connected with that node with at least one ray. The priorities of the affected candidates have to be reevaluated and their position in the priority queue has to be updated. In the extreme case, a split of a single view space or object space node can affect all of the split candidates from the other domain.

In order to cope with this situation, we exploit the following observation: when performing a split, the priority of other candidates in the queue remains valid if they are from the same domain as the split. This follows from the fact that the cost evaluation of disjoint nodes of the same domain is independent. We use this observation to reduce the number of recomputations of split candidate priorities. The optimized algorithm maintains separate priority queues for split candidates of view space and object space domains and proceeds as follows: Initially we take the split candidate with the highest priority by comparing the fronts of the view space and object space priority queues. We take at least n_{min} splits from the same domain *without reevaluation* of split candidate priorities. Then we compare the priority of the current split candidate with the best split of the other domain. If the priority of the current split is lower (condition 1), we recalculate the priorities of the candidates and decide whether to switch the domains. Otherwise we continue subdividing in the current domain until condition 1 is met or we reach n_{max} splits (condition 2). n_{max} is a safety criterion ensuring that reevaluation is made with sufficient frequency. As a result the number of steps without reevaluating the split candidate priorities is in the range $< n_{min}, n_{max} >$.

Even when processing the splitting planes in batches, the update of all affected split candidates per batch is expensive. The deeper the subdivision, the more candidates have to be updated. To further reduce the number of recomputations of priorities, we only update a subset of split candidates. This subset is chosen randomly from the set of affected candidates.

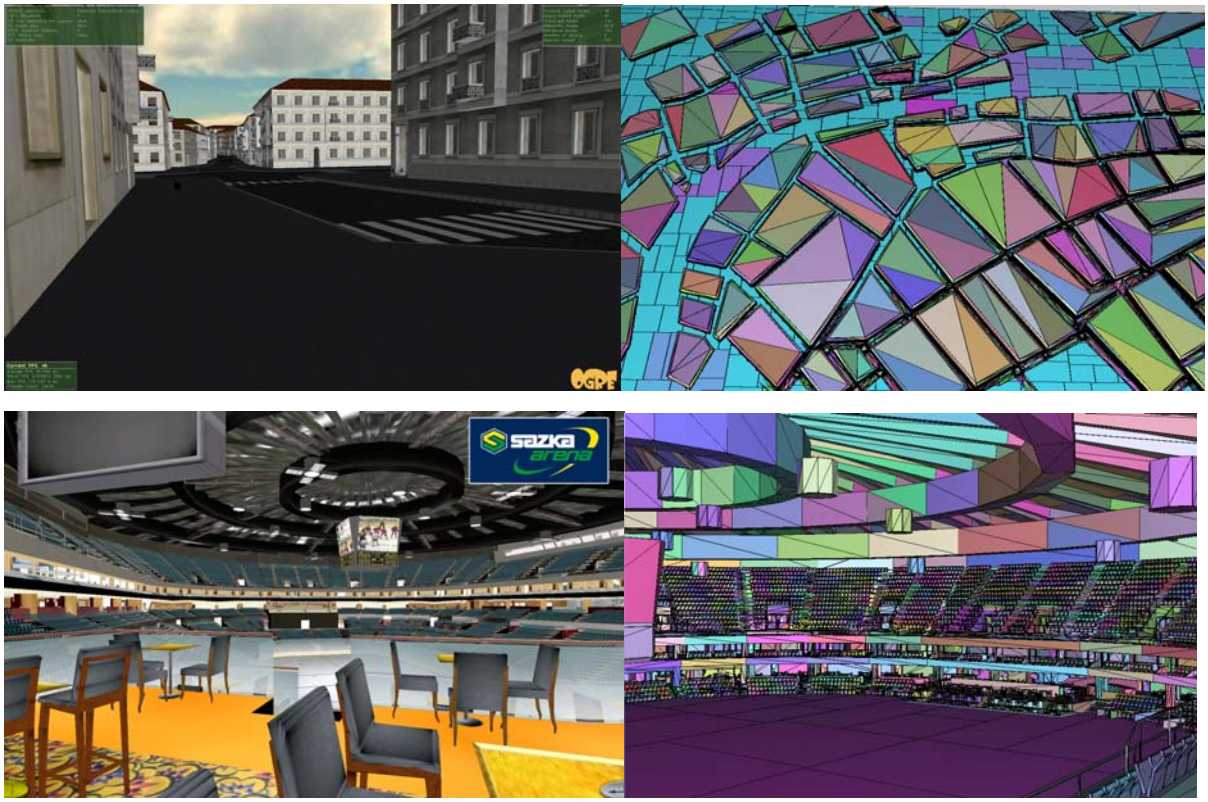

Figure 12, Snapshot of Vienna scene (upper left) and visualization of the view / object space partition (upper right). Snapshot of Arena scene (lower left) and visualization of the view / object space partition (lower right). Each colored patch represents one object from the PVS. We show a cut through the 3d view space subdivision. The view cells are colored from blue (=low render cost) to magenta (=high render cost) The Arena scene is a courtesy of Digital Media Production a.s.

3.3. PVS COMPUTATION USING PROGRESSIVE VISIBILITY SAMPLING

This module deals with calculating visibility in a preprocess. It determines a potentially visible set (PVS) of objects for every view cell. In runtime the view cell containing the current viewpoint is located and only the objects contained in the associated PVS have to be rendered.

Visibility is an important problem in computer graphics, with many applications in architectural walkthroughs, computer games, urban planning, traffic simulation and others. One typical way to compute visibility is as a preprocess to a time-critical application like a walkthrough or a computer game. This has the advantage that invisible stationary objects can be eliminated at practically no runtime cost. Visibility preprocessing algorithms subdivide the view space into view cells and compute a potentially visible set (PVS) of objects for each view cell. At runtime, the view cell containing the current viewpoint is located and only the objects contained in the associated PVS have to be rendered.

The standard scheme of a visibility preprocessing algorithm is:

```
for each view cell v
  compute all visibility interactions in v
  create 1 PVS for v
```

This concept is straightforward and it is known from the early 90s. It is noteworthy that visibility calculation times for such algorithms are usually quoted in time per view cell---for a good reason: the high calculation times for whole scenes make visibility preprocessing cumbersome and preclude its use for anything else than the final delivery of a product.

In the progressive visibility sampling module we implemented a visibility preprocessing framework based on which redefines the classical preprocessing scheme. Instead of solving visibility sequentially for individual view cells, we progressively compute global visibility, i.e. visibility for all view cells. This basically corresponds to the following scheme:

```
while(!terminate)
  compute 1 visibility interaction
  augment all affected PVSs
```

This has two key advantages: 1) The solution is truly progressive: a coarse global visibility solution is obtained within a few seconds. As time passes, more visible objects are discovered. This is important as it opens up completely new fields for visibility processing. For example, the algorithm can be used as a fast preview tool for visibility analysis and to find visibility hotspots in large scale modeling. 2) The solution exploits the spatial coherence of visibility since each visibility interaction contributes to many view cells. Thus, the convergence of the solution is accelerated by at least one order of magnitude.

A global visibility sampling solution entails a number of issues, foremost the question how samples should be chosen. The main problem of visibility sampling is that visibility interactions do not appear uniformly in ray space: for a particular view cell, far away parts of the scene require significantly more samples than nearby parts. This makes uniform sampling strategies inefficient, as they have to sample the whole ray space at the highest necessary sampling density.

It has been shown that for a single view cell and triangle-based scenes, deterministic sampling strategies are a good way to efficiently choose samples that contribute to visibility. However, working on a triangle level is infeasible for simultaneous global PVS computation for large scenes with many view cells and millions of triangles (e.g., a scene with 5,000 view cells and about 100,000 visible triangles per view cell would require 2GB of memory). On the other hand, general, non-triangle objects are difficult to handle for deterministic sampling strategies due to the computational complexity of e.g. silhouette finding.

Another problem of deterministic sampling is that it contradicts the idea of progressive evaluation. For example, recursively creating samples at the silhouettes of each encountered object quickly leads to a depth-first exploration of local ray space neighborhoods. As an extreme example, consider a view cell above a flat terrain. The first ray intersection will induce a 'visibility flood fill' of the whole terrain. Such a strategy is not progressive even for a single view cell. Furthermore, it requires a significant traversal state per view cell, so that the computation cannot easily be distributed among different view cells. Basically, one view cell has to be completed before starting the next one. In this paper, we present probabilistic sampling strategies based on ray mutations that work for arbitrary objects and take into account the non-uniformity of ray space in a similar way as deterministic strategies, while at the same time preserving the progressivity of the evaluation.

3.3.1. Algorithm Overview

The Progressive Global Visibility (PGV) algorithm is built on the idea of using *sampling* to determine visibility. At the core of our algorithm is a *global visibility sampler*, which progressively casts bidirectional rays to determine maximal free line segments in the scene, and then determines their contribution to all view cells. Thus a single visibility sample can contribute to many view cells and therefore exploit visibility coherence.

The second contribution deals with how these samples are generated. The 4D sampling domain is too large to quickly capture all important rays by regular sampling. Therefore we use different heuristical distributions which are combined in an adaptive mixture distribution that takes into account their success in discovering new PVS entries.

We introduce a number of ray distributions that are suitable for a global visibility algorithm. *Stationary distributions* (Section 3.3.2.3) sweep visibility globally and seed the *mutation-based distributions* (Section 3.3.2.4), which focus the sampling at places of visibility changes and allow adapting the sampling rate to the distance of visible objects.

Finally, we present the concept of *visibility filtering*, which counteracts errors due to undersampling in early stages of the algorithm by including in the PVS also objects that are likely to be visible based on the sampling density.

3.3.2. Progressive Global Visibility Sampling

3.3.2.1. Global Visibility Sampling

Visibility sampling relies on *ray casting* to obtain visibility information. One main reason for the efficiency of our approach is the use of spatial coherence in visibility. In contrast to a per-view cell algorithm, where each ray can only contribute to one single view cell, we determine the contribution of each ray to all view cells it encounters. For this, we use *visibility samples* created from rays.

Visibility sample definition

In visibility sampling algorithms, a sample ray r is normally defined starts in a view cell (x_r *in* v) and with a direction d_r. A visibility contribution is then determined by shooting the ray using a standard ray caster and determining the closest object $h(r)$. We define the contribution of a ray as

$$\begin{aligned} H(r) &= \{h(r)\} \quad \text{if } h(r) \in O \text{ (object hit)} \\ &= \{\} \quad \text{otherwise (no object hit)} \end{aligned}$$

In global visibility, however, we are not only interested in the visibility contribution of the hit object to the view cell at the ray origin, but to all view cells pierced by the ray along an unobstructed path from the hitpoint. This is equivalent to the *maximal free line segment* defined by r. To obtain this line segment, we not only shoot the original ray, but also the ray in the opposite direction, i.e., $-r = (x_r, -d_r)$, and obtain a second hit $h(-r)$. We thus obtain a line segment s associated with zero, one or two visible objects at its endpoints: $H(s) = H(r) \cup H(-r)$. A line segment with one or two visible objects $|H(s)|>0$ is a *valid sample*.

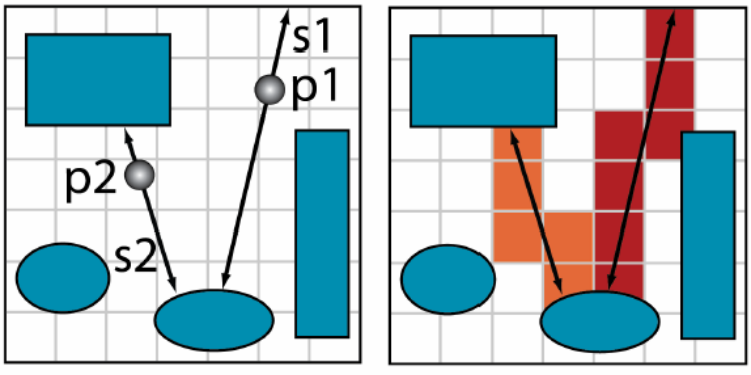

Figure 13, Left: This figure shows the creation of visibility samples. The scene objects are shown in blue. A sample point (see *p1* and *p2*) is generated together with a direction vector. A ray is cast from the sample point in the generated direction as well as into the opposite direction. Right: A ray associates all its points with visibility of object(s) on its endpoints. These objects are added to all PVSs of view cells pierced by the ray.

Updating view cells.

The view cells affected by a valid sample are determined by intersecting the line segment with the data structure containing the view cells. The objects associated with the sample are then added to *all* view cells pierced by the line segment.

Sample contribution

During this process, the *contribution* of the sample is evaluated in order to distinguish between samples which give us valuable information about visibility and samples which either hit no object or hit objects already discovered as visible. This measure will be used to drive the sampling process (Section 3.3.2.2).

The local contribution of the sample to a particular view cell equals the number of objects associated with the sample that are added to the PVS of the view cell (and have not already been included in the PVS):

$$\bar{c}(s,v) = |H(s) \setminus PVS(v)|$$

The total contribution c(r) of the sample is simply the sum of contributions to all view cells *V(r)* which are pierced by the sample:

$$c(r) = \sum_{v \in V(s)} \bar{c}(s,v)$$

3.3.2.2. Adaptive Mixture Distribution

In addition to the explicit use of spatial visibility coherence, another major source of efficiency of our global visibility algorithm is the ability to adapt to the visibility structure in the scene. This is done in two ways: first, by choosing sample distributions according to previous visibility contributions, and second by using a ray mutation strategy which places samples near visibility events and adapts to the required sampling density in ray space.

We have found that no single sampling strategy is efficient for all types of scenes, as the efficiency of a sampling strategy depends on the scene properties and its visibility characteristics. We therefore employ a probabilistic approach based on a distribution mixture: we allow the use of several different distributions to generate visibility samples. The success of previous samples generated by each distribution is used to drive its selection probability, thus automatically adapting to the scene visibility properties.

More specifically, the contribution *C(D)* of a sample distribution *D* at a specific point in time is defined as the sum of sample contributions *c(s)* of all samples *s* in a reference set *S(D)*. The reference set is typically the set of rays generated by the distribution *D* in a certain time window:

$$C(D) = \sum_{s \in S(D)} c(s).$$

To measure the success of a distribution in discovering visibility, we define its *weight w(D)* as the contribution with respect to the reference set:

$$w(D) = C(D)/|S(D)|$$

The probability for drawing a sample from distribution D_i is then simply

$$p(D_i) = \frac{w(D_i)}{\sum_D w(D)}.$$

The following two subsections describe the distributions which we use n the algorithm. Three of these distributions are stationary, while two mutation-based strategies adapt to the actual visibility contributions.

3.3.2.3. Stationary Ray Distributions

A distribution is stationary if each sample is independent of previous ones. These distributions are used to provide an initial efficient covering of ray space and to seed the more adaptive mutation-based strategies described below. The stationary distributions explore the whole ray space and therefore guarantee the progressivity of the algorithm.

In contrast to per-view cell visibility, there is no obvious 'uniform' sample distribution. Instead, the best strategy to discover new visible objects depends on the visibility configuration of the scene. We list three distributions which we have found to work well. All rely on low-discrepancy series like the Halton sequence to generate samples. We use independent random variables in the range *[0,1)* denoted φ_1, φ_2 etc.

View space-direction distribution.

This distribution makes sure that all view cells are sampled in all directions. We generate a random point *x* in view space using a uniform distribution and a random direction *d* using a uniform distribution in the directional space with spherical coordinates:

$$\phi = 2\pi\psi_1, \quad \theta = \arccos 1 - 2\psi_2$$

Object surface-direction distribution.

This distribution makes sure that all objects are sampled in all directions (note that this can be inefficient if the view space does not include all of object space). We generate a random point *x* on the surface of a randomly chosen object, and a random direction *d* using a cosine-weighted uniform distribution on the hemisphere erected over the tangent plane of *x*. The spherical coordinates of *d* are:

$$\phi = 2\pi\psi_1, \quad \theta = \arccos\sqrt{\psi}$$

Two-point distribution.

This distribution focuses on the most probable visibility interactions between objects and view cells. We generate a random point *o* as in he object surface-direction distribution, and a random point *v* as in he view space-direction distribution. The ray generating the visibility sample is then *r=(v, s-v)*. The most important feature of he two-point distribution is that it adapts to the shape of the scene. or example, typical urban scenes are much wider than they are high. his fact is taken into account in the two-point distribution and most samples are cast in a roughly horizontal direction. For this reason, he two-point distribution is typically the most successful stationary sampling strategy.

For positions generated in view space, we also allow the following modification: instead of using a uniform distribution, the distribution is chosen according to a user-supplied density function. In an interactive application, or example, this can be used to focus visibility computations (carried out in the background) on view cells in the vicinity of the observer by choosing a density function that falls off with increasing distance to the current viewpoint. Another option is to let the level designer designate regions of higher importance.

3.3.2.4. Mutation-Based Distributions

Even using an optimal mix of stationary sample distributions, the size of ray space that needs to be sampled is prohibitively high. The required sampling density is highly non-uniform: for a particular view cell, far away regions need o be sampled more densely than near regions. Furthermore, it has been shown that the efficiency of visibility sampling can be vastly improved by trying to sample near possible changes in visibility. We exploit these two observations using mutation-based sampling distributions: a *two-point mutation* distribution and a *silhouette mutation* distribution.

Mutation candidate maintenance.

Both strategies work on the following principle: During the sampling process, each sample with on-zero contribution $c(s)$ (regardless of which distribution generated it) is stored as a candidate for mutation. In case the visibility sample has two objects associated with it ($|H(s)|=2$), and the other object actually generated a contribution, we generate two mutation candidates from it, as we distinguish between the segment termination point (where the object under consideration is hit) and the segment origin (either the intersection with an object or a view cell). These two points and the object id of the object are stored with the sample.

All mutation candidates for a strategy selected for storage are collected in a buffer. When a new sample is requested from a mutation-based strategy, a candidate is chosen from the buffer. However, instead of choosing randomly from the buffer, we observe that he history of mutations conveys important information about the possible importance of a candidate: If a candidate has received a large umber of mutations already, it is likely that its neighborhood is already well explored. On the other hand, new mutation candidates have not received any mutations so far, and especially after a certain run-in time, such new mutation candidates represent more ‘difficult’ cases of visibility. We therefore use the mutation count generated from this candidate as a value to sort the buffer, and always elect the candidate with the lowest value. If the buffer is full, the element with the highest value is dropped.

Two-point mutation.

The aim of the two-point mutation is to adapt the sampling rate of visibility to the complexity of ray space. More specifically, given a segment $s=(s_o, s_t)$, the segment termination point s_t is mutated o as to discover nearby *objects*, while the segment origin s_o s mutated so as to discover nearby *view cells* (note that it is entirely due to the global nature of the algorithm that a strategy for discovering new view cells is possible at all).

Let $h(x)$ be the object or view cell (in case the reverse ray generating the segment didn't hit an object) associated with point x in (s_o, s_t), and $r(o)$ the radius of the bounding volume of object or view cell o. Then we construct a plane perpendicular to the segment, and mutate x by drawing a new point x' from a two-dimensional Gaussian distribution on this plane centered at x with standard deviation $\sigma = r(h(x))$ (see Figure 14, left).

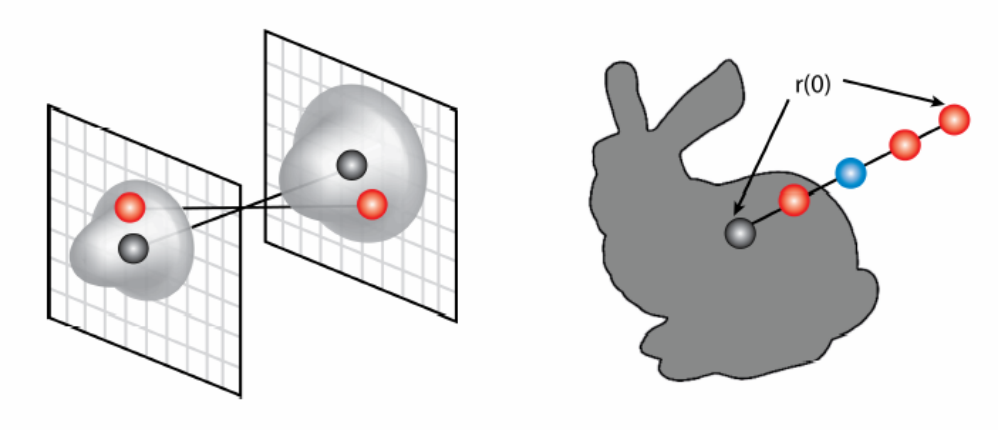

Figure 14, Left: The two-point mutation moves both start and endpoints according to Gaussian distributions. Right: Silhouette search.

Note that for both mutation-based strategies, once we have obtained the new segment (s'_o, s'_t), a visibility sample is generated from this segment by creating a forward ray r' (together with reverse ray $-r'$ with $x'_r = (s'_o, s'_t)/2$ and $d'_r = s'_t - s'_o$. The new ray origin is chosen at the center of the new segment so as to avoid local occlusion at the start or termination points of the segment.

Silhouette mutation.

The silhouette mutation adapts the deterministic adaptive border sampling strategy proposed by Wonka et al. for a progressive setting. This strategy places samples near the silhouettes of newly discovered objects, where changes in visibility are most likely to occur. However, deterministically sampling the whole silhouette of an object would not only be computationally expensive, but it would also quickly saturate the list of mutation candidates.

Therefore we chose a probabilistic approach that randomly selects *one* silhouette point. As in the two-point mutation, a plane perpendicular to the segment is placed at s_t. On this plane, we choose a random direction d. Then we shoot 'discovery rays' along the segment $(s_t, r(o) \cdot d)$. The first ray that does not intersect the object is chosen as a silhouette ray and returned as mutation point s'_t to construct a new visibility sample (see Figure 14, right).

Note that for most ray tracers, these discovery rays can be evaluated much faster than normal samples. First, the region of interest can be restricted to the bounding volume of the object so that the ray does not need to be intersected with the whole scene. Second, the segment origin remains fixed for all rays, so that ray packet optimizations can be exploited. For example, shooting packets of 4 rays, a quaternary search of depth 3 gives an accuracy of $r(o)/125$.

Important visibility events appear at possible *depth discontinuities* discovered by the silhouette sample. For this, we adapt the reverse sampling strategy described by Wonka et al. When the

silhouette sample $r=(s_o, s'_t-s_o)$ is cast, a depth discontinuity is reported if the distance between mutated segment endpoint s'_t and the actual hitpoint $h(r)$ is larger than 3 times the original object radius:

$$s'_t - h(r) > 3r(o)$$

This margin is intended to avoid situations where a closer hitpoint on the same object could be interpreted as a depth discontinuity. Now instead of looking for the exact depth discontinuity as in reverse sampling, we do a new silhouette mutation as described above using the segment $(s'_t, h(r))$, and store the resulting ray along with the original silhouette ray.

3.3.2.5. Putting It All Together

The complete Progressive Global Visibility algorithm works in a loop as follows:

```
while (!terminate)
{
  select distribution
  draw a sample from selected distribution
  cast forward and backward rays
  if (hit)
  {
    update view cells
    if (contribution > 0)
      store sample as mutation candidate
  }
  update distribution probabilities
}
```

Termination.

One issue that deserves attention is when to terminate calculation. For a per-view cell sampling algorithm, the quality and running time of the complete solution depends significantly on setting the termination criterion. Interestingly, in a *global progressive* visibility algorithm the termination criterion is not critical, as the user can inspect the current visibility solution at any time. For offline use, a termination criterion can be set on the maximum PVS increase for a view cell for a certain number of rays. E.g., when during 5M rays no PVS increases by more than 5 objects, the algorithm will terminate.

Loop optimization.

In practice the loop is not carried out for individual rays, but for larger batches of rays (e.g., 1M). This allows reordering the samples so as to achieve better coherence for the ray caster. In addition, if the window used for calculating ray distributions is chosen to be exactly one batch, the step of updating the distribution probabilities *C(D)* (see Section 3.3.2.2) needs to be done only once per batch.

3.3.3. Visibility Filtering

While Progressive Global Visibility is an aggressive algorithm (i.e., the PVSs only contain a subset of the exact visible set), the sampling strategies described in the previous section aim to approach the exact solution as quickly as possible. However, especially in the initial phases of the algorithm, visible errors will appear.

To cope with this problem, we introduce the so-called *visibility filter*, which extends the computed PVSs by additional objects which are likely to be visible. The main novelty is that we take into account the visibility error expected from the sampling density. This means that PVSs for regions which have been sampled densely will not be extended significantly, whereas undersampled regions will have more objects added. The visibility filter fills gaps in 'visible fronts' that appear when the sampling rate in a region is lower than the object density. It can not discover objects that are visible in an isolated manner (i.e., objects smaller than the sampling density that have no visible neighbors).

The visibility filter can be applied as a postprocess to all PVSs after running the main algorithm, or it can be evaluated lazily during walkthrough for the current view cell. We present an o*bject space filter*, which adds objects in proximity of already visible objects, and a *view space filter*, which merges visible objects from neighboring view cells.

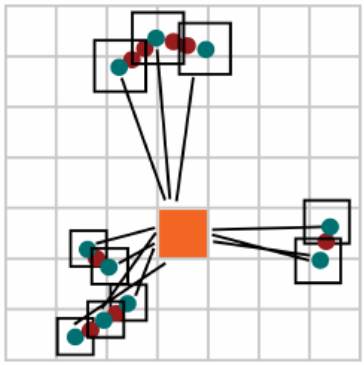

Figure 15, Illustration of the object space visibility filter. Objects originally in the PVS are shown in blue. The extended bounding volumes are shown in black. Objects added to the PVS by the filter are shown in red.

Object space filter.

The object space filter works on a view cell *v* and its potentially visible set PVS_v. For each object o in PVS_v, the amount of *extension e(o)* is calculated as the estimated distance to a visibility sample in the vicinity of the object. The object space filter is then applied by extending the bounding volume of each object *o* by *e(o)* (e.g., the radius of a bounding volume or the axes of a bounding box), and adding to PVS_v all scene objects that intersect one of the extended bounding volumes (see Figure 15). The extension *e(o)* can be calculated either from the density of all visibility samples intersecting the view cell (global extension $e_{g(o)}$), or from the density of samples that hit the object (local extension $e_{l(o)}$).

The *global extension* estimation assumes that n_v visibility samples are uniformly distributed on a sphere of radius *d(o)*, which is the distance of the object from the view cell. As extension we use half the approximated distance between two neighboring samples on this sphere:

$$e_g(o) = \frac{2d(o)}{\sqrt{n_v}}.$$

The *local extension* assumes that the $n_{v(o)}$ visibility samples which hit the object are uniformly distributed on a disk of radius *r(o)*, which is the radius of the object bounding volume. As extension we use half the approximated distance between two neighboring samples on this disk:

$$e_l(o) = \frac{r(o)}{\sqrt{n_v(o)}}.$$

Note that both estimations are not accurate: the global estimation ignores that rays due to the mutation-based strategy are not distributed uniformly, while the local estimation ignores that parts of the object may be occluded from the view cell, leading to an overestimation of e_l. In practice we therefore choose the minimum of the two estimations, and allow the user to increase or decrease the filter size with a constant *k* for more conservative or more aggressive results: $e(o)=k \cdot min(e_{g(o)},e_{l(o)})$.

View space filter.

The view space filter is useful only if the size of the view cells is relatively small (i.e., comparable to size of the objects). This filter is very simple: it merges the PVS of the given view cell with the PVSs of all neighboring view cells.

3.3.4. Implementation

We have implemented the PVS computation module as a multithreaded application . Initially the scene and the view cells are loaded into memory and the initial set of rays is cast. Casting the initial ray set is performed on the CPU and takes a few seconds. After propagating the ray visibility contributions to the view cells the scene can already be rendered using the view cell based PVS. At this stage visible errors appear due to undersampling however the most of the visible objects is already rendered correctly. The user can freely walk in the scene and while the visibility processing thread updates the PVSs. Depending on the scene it takes few tens of seconds to few minutes till the PVSs capture enough objects so that most frames contain no visible errors.

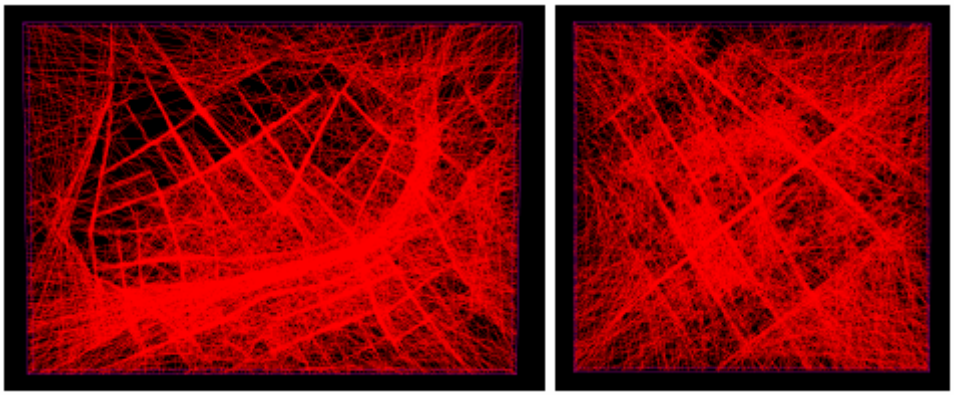

Figure 7: Subset of rays generated by the PGV algorithm in one batch after about 20M samples have been cast. (left) Vienna (right) Pompeii.

3.4. PVS COMPUTATION USING GUIDED VISIBILITY SAMPLING

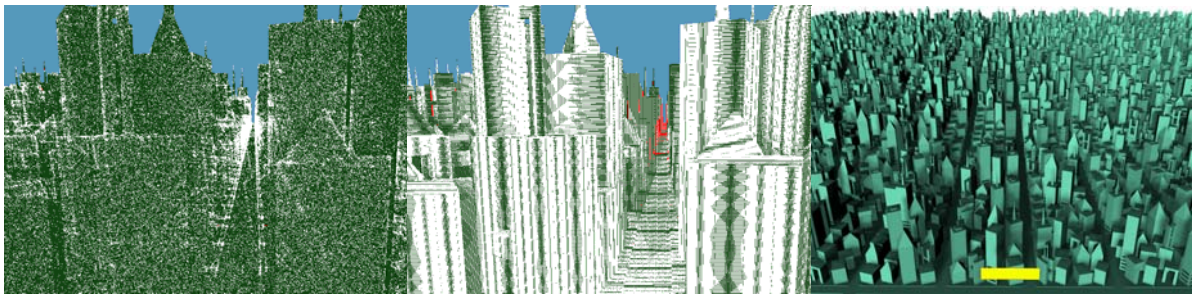

Figure 16, Visualization of sampling strategies (white pixels show a subset of the actual samples, missed geometry is marked red). Left: An urban input scene and a view cell (in yellow) for visibility sampling. Middle: Previous visibility sampling algorithms repeatedly sample the same triangles in the foreground while missing many smaller triangles and distant geometry. Right: Our solution is guided by scene visibility and therefore quickly finds most visible triangles while requiring drastically fewer samples than previous methods.

This method is similar to the method presented in Section 3.3 in that it calculates a visibility solution for a set of view cells. The difference is that the underlying algorithm it is able to provide a triangle-based PVS and therefore the solution is more accurate. The sampling method is also more deterministic and is capable of providing PVSs that are comparable in accuracy to exact methods as evidenced later. Obviously, this comes at the cost of higher computation time than progressive global visibility. Since Guided Visibility Sampling is much more efficient and robust than exact visibility methods, and provides comparable accuracy, we have decided to use this method instead of a real exact solution to provide the most accurate PVS computation.

This method addresses the problem of computing the triangles visible from a region in space. The proposed aggressive visibility solution is based on stochastic ray shooting and can take any triangular model as input. We do not rely on connectivity information, volumetric occluders, or the availability of large occluders, and can therefore process any given input scene. The proposed algorithm is practically memoryless, thereby alleviating the large memory consumption problems prevalent in several previous algorithms. The strategy of our algorithm is to use ray mutations in ray space to cast rays that are likely to sample new triangles. Our algorithm improves the sampling efficiency of previous work by over two orders of magnitude.

While not being globally progressive like the method introduced in the last section (we solve one view cell at a time), this method aims for exactness. This method is much faster and robust than so called “exact” geometrical visibility solver. We found out that the new method even provides more accurate results than the geometrical methods, as they all suffer from robustness issues. In case that we require an exact visibility solution, we choose this sampling method over a geometrical visibility solver. For these practical reasons we did not see the benefit in additionally providing a geometrical method

3.4.1. Introduction

Visibility is a fundamental problem in computer graphics: visibility computations are necessary for occlusion culling, shadow generation, inside-outside classifications, image-based rendering, motion

planning, and navigation, to name just a few examples. While visibility from a single viewpoint can be calculated quite easily, many applications require the potentially visible set (PVS) for a region in space, which is, unfortunately, much more complicated. A number of excellent from-region visibility algorithms exist, but most of them are only applicable to a limited range of scenes, require complex computations, and sometimes significant amounts of memory. Therefore, sampling-based solutions have become very popular for practical applications due to their robustness, general applicability, and ease of implementation. In this paper we will improve upon previous sampling-based algorithms by significantly improving the sampling efficiency, i.e., the number of samples required to detect a certain set of visible polygons. To motivate our design choices, we will look at two key aspects of any visibility algorithm: the behavior of the algorithm in ray space, and the data structure used to store and acquire visibility information.

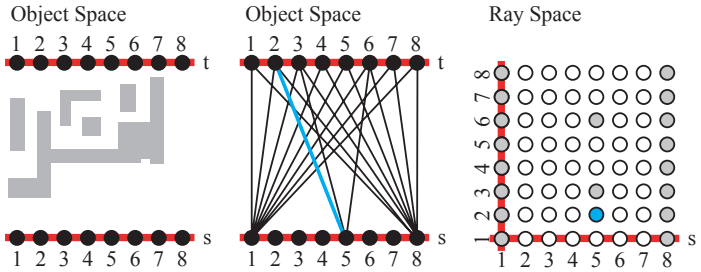

Figure 17, Sampling in object and ray space. Left: a scene with a set of objects. A view cell is shown as a line segment parameterized with *s*. We are interested in all rays that intersect the view cell and a second line segment parameterized with *t*. Middle: Shows a subset of the possible rays. One ray is highlighted in blue. Right: A depiction of the discrete ray space. Any ray in the middle figure corresponds to a point in ray space. The blue point corresponds to the blue ray in the middle figure.

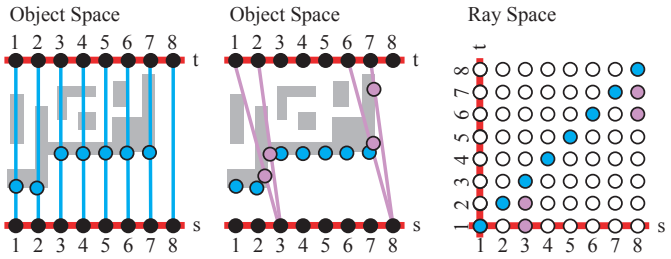

Figure 18, Left: The scene sampled orthogonally. Middle: Additional samples to capture oblique surfaces. Right: The rays used to sample the scene are shown in corresponding colors.

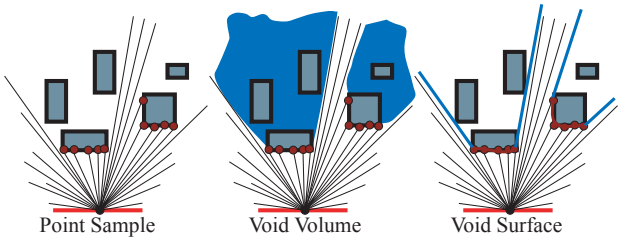

Figure 19: Representing visibility from a single point. Left: independent samples. Middle: the void volume. Right: the void surface.

Figure 17 illustrates the concept of ray space in 2D. Given a view cell, shown as edge parameterized with *s*, and a scene with objects shown in grey, we can compute visibility by considering all rays from the view cell to a plane behind the scene, parameterized with *t*. For a 2D scene, this is a 2D set of rays; for a 3D scene this is a 4D set of rays. If this set of rays is sampled densely enough, we will have a good visibility solution. The inefficiency of a pure regular sampling approach as shown in Figure 17 is that the same surfaces are sampled over and over again (note that the definition of regular depends on the parameterization of ray space!). Therefore, it would be beneficial if we could only sample areas that have not been sampled before. This is shown in Figure 18, where after an initial orthogonal sampling, only few additional rays are needed to find all visible surfaces. In total, little more than a 1D subspace of the 2D ray space needs to be explored in this example. This is due to the spatial coherence of visibility. In this paper, we exploit this coherence: starting from stochastically sampled points, we grow lower-dimensional subspaces of ray space using the newly introduced strategies of *adaptive border sampling* and *reverse sampling*, which are guided by the properties of scene visibility.
The second key aspect of a visibility algorithm is what data structure is used to store visibility information. The most complete, but also complex, way is to store 4D ray space. For large scenes, this entails prohibitive levels of memory consumption. Conservative algorithms often store the shadow volume, whereas sampling algorithms use the volume of 3D space that has not been sampled yet (the so-called void volume, Figure 19); but these data structures still require several times the memory taken by the scene description itself. Alternatively, the boundary of the void volume (the void surface) can be used, which is easy to sample from one point in space, but difficult to manipulate. In this paper, we do not store visibility information beyond the PVS at all, relying on our new reverse sampling approach to penetrate the void surface based on the current sample only. The key contribution of this paper is an intelligent sampling algorithm that drastically improves the performance of previous sampling approaches by combining random sampling with deterministic exploration phases. The algorithm requires little memory, is simple to implement, accepts any triangular test scene as input, and can be used as a general purpose visibility tool.

3.4.2. Overview

3.4.2.1. Problem Statement

We consider visibility problems that are posed as follows: As first input we take a three-dimensional scene consisting of a set of triangles, *TS*. We do not rely on connectivity information, volumetric objects, or large polygons as potential occluders (such a set of triangles is often called triangle soup). As second input we consider a subset of ray space Ω, usually defined by the rays emanating within a 3D polyhedron called *view cell* and intersecting the bounding box of the scene. A ray can be defined by a starting point and a direction. Using *TS* and Ω, we can define a visibility function $v : \Omega \rightarrow TS$, so that each ray in Ω maps to the triangle in *TS* that it intersects first. The exact solution of the visibility problem is the range of this function, $v(\Omega) \subseteq TS$, also called exact visible set EVS. Our algorithm is *aggressive*, i.e., it calculates a potentially visible set $PVS \subseteq EVS$. Our algorithm can be used to solve the visibility problem in different applications (see Section 3.4.4.6). A *usage scenario* to keep in mind for the following exposition is a visibility preprocessing system for real-time rendering: the *view space* (set of possible observer locations) is partitioned into view cells. In a *preprocessing* step, our algorithm is used to calculate and store a PVS for each view cell (note that only its boundary polygons are taken into account, since any ray leaving the view cell can be represented by a ray on the boundary). At runtime, the view cell corresponding to the current observer location is determined, and

only the objects in the associated PVS are sent to the graphics hardware, leading to significant savings in rendering time.

3.4.2.2. Algorithm Overview

The algorithms in this paper are based on ray shooting and assume the capability to trace a ray x and compute its first intersection with a scene triangle $t \subseteq TS$, i.e., to compute the triangle $t = v(x)$ (fast ray tracers include OpenRT and the recently presented MLRTA).
The idea of a sampling solution is to select a sequence of rays $X = x_i$, trace the ray and add the triangle $v(x_i)$ to the visibility set *PVS*. In this paper, we will address the problem on how to sample *efficiently*, that is how to improve the chances of finding new triangles. We will start with one of the most popular sampling strategies, random sampling (section 3.4.3.1). Then we will show how to use visibility information from previous samples to construct intelligent sampling strategies based on ray mutation to complement random sampling: *Adaptive Border Sampling* is an algorithm to quickly find nearby triangles by sampling along the borders of triangles previously found to be visible (Section 3.4.3.2). *Reverse Sampling* is an algorithm to sample into regions in space that are likely to be near the boundaries of visible and invisible space, i.e., the void surface (Section 3.4.3.3).
In Section 3.4.3.4, we will show how to combine the different sampling algorithms in order to obtain *guided visibility sampling*, a complete hybrid random and deterministic sampling algorithm. The algorithm is called guided because both sampling strategies are guided by visibility information in the scene (see Section 3.4.4.4 for a more detailed discussion).

3.4.3. Visibility Sampling

All rays in the scene form a 5D space. A ray x has a starting point x_p (3D) and a direction x_{dir} (2D). A typical visibility query is to give a region R in 3D space and ask what is visible along the rays leaving the region (view cell). While this defines a 5D set of rays, we only need to consider a 4D set of rays in practice; the rays starting at the boundary δR of the viewing region. Additionally, all triangles intersecting R are classified as visible.

3.4.3.1. Random Sample Generator

The random (or pseudo-random) sampling algorithm selects a sequence of random samples $X = xi$ from the scene. The probability distribution for each new sample $p(xi)$ is independent of all previous samples $x_1, \ldots, x_{n-1}$. The question of sampling uniformity for random sampling has been explored in the context of form factor computation. We sample the position and ray direction uniformly using the following formulae:

$u = \xi 1$, $v = \xi 2$, $\varphi = 2\pi\xi 3$, $\theta = \arcsin \xi 4$,

where the ξi are independent Halton sequences, and (u, v) are the normalized coordinates on a view cell face. While random sampling alone suffers from similar inefficiencies as regular sampling, it will be used to seed the more efficient strategies described next.

3.4.3.2. Adaptive Border Sampling

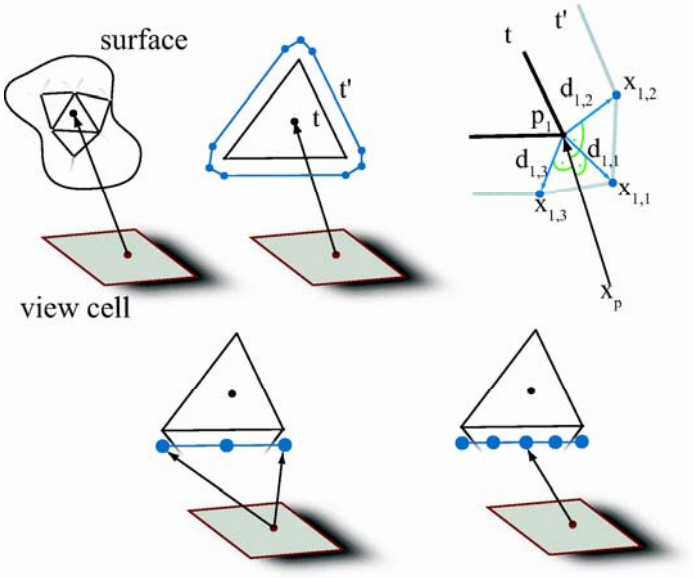

Figure 20, Adaptive border sampling: Top: If we hit a new surface, we sample nearby points on the border polygon *t'*. Bottom: Adaptive subdivision of an edge of *t'*.

This sampling algorithm is a deterministic ray mutation strategy that covers most of the ground work to make our system successful. This strategy leaves the ray starting point x_p on the view cell fixed while covering adjacent triangles in object space, practically constructing a local visibility map from the selected view cell point.
The key idea of this sampling strategy is that it adapts the sampling rate to the geometric detail of the surface (see Figure 20). Therefore, it is unlikely that subpixel triangles are missed, which is a problem for methods that sample objects regularly. The method performs especially well for the most frequent case of a connected mesh, but does not assume or use any connectivity information. The connected regions are discovered in the random sampling step (therefore, scenes with many small disconnected meshes like trees remain a challenge for the approach). The algorithm proceeds as follows. If a triangle $t = (p_1, p_2, p_3)$ is hit for the first time by a sample ray $x = (x_p, x_{dir})$, we enlarge t by a small amount to obtain an enlarged polygon t', and adaptively sample along its edges (Figure 20). For each edge, we use two rays x_l and x_r, and the corresponding samples $hit(x_l)$ and $hit(x_r)$ in world space. If the rays x_l and x_r hit different triangles, we recursively subdivide the edge, up to given threshold. At this point, we also detect depth discontinuities between the new samples and the original sample on the triangle, which is already a part of reverse sampling as described in the next section.

The actual method used for *border enlargement* deserves attention. In order not to miss any adjacent triangles, the border polygon t' should be as tight as possible. On the other hand, if it is too tight, t will be hit again due to the numerical precision of ray shooting. If the enlargement were done in object space, this would happen for near edge-on or very distant triangles. We therefore enlarge t in ray space by rotating rays to the vertices of t to their new positions on t' by a small angle. This is more robust because it depends neither on the distance of the triangle nor on its orientation, but only on the numerical precision of the ray representation. In practice, this means that for each vertex, the new vertices are put on a plane perpendicular to the ray. The shape of t' is chosen so that the ray space distance to t is fairly constant. This is not possible with only 3 vertices, since sliver triangles would lead to singularities. We therefore chose t' to be a polygon of 9 vertices. For each vertex p_i of t, three

vertices *pi, j* on *t´* are generated. Two vertices are generated each on a vector $d_{i,j}$ perpendicular to the ray and to one of the adjacent edges, respectively. The third is the midpoint of the other two, pushed away from *t* along the angle bisector $d_{i,i}$:

$d_{i,i+1} = N((p_i - x_p) \times (p_{i+1} - p_i))$
$d_{i,i-1} = N((p_i - x_p) \times (p_i - p_{i-1}))$
$d_{i,i} = N(d_{i,i-1} + d_{i,i+1})$ *if* $d_{i,i-1} \cdot d_{i,i+1} > 0$, *else:*
$N((pi - x_p) \times d_{i,i-1} + d_{i,i+1} \times (p_i - x_p))$
$p_{i,j} = p_i + \varepsilon \cdot |p_i - x_p| \cdot d_{i,j}$

where $N(v)$ is the vector normalization operator. *di,i* is chosen to be numerically robust. For backfacing triangles, the *di,j* need to be inverted. Adaptive border sampling efficiently explores connected visible areas of the input model from a single viewpoint along a 1D curve in ray space. However, it cannot penetrate into gaps. This is handled by reverse sampling.

3.4.3.3. Reverse Sampling

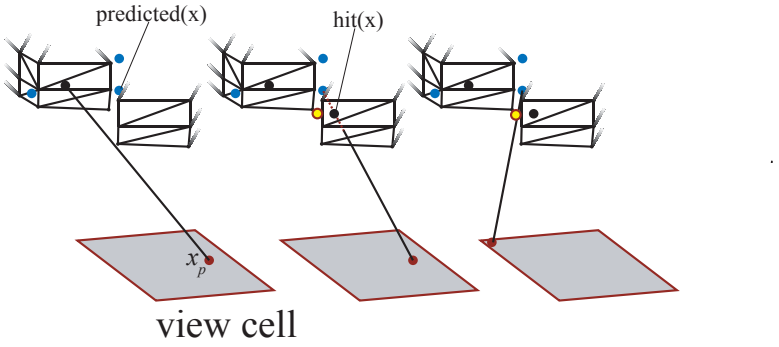

Figure 21, Reverse Sampling: Left: initial hit on triangle *t*. Middle: the new ray to *predicted(x)* is blocked by a much closer triangle *t'*. Right: Reverse sampling mutates the starting point on the view cell so that the ray passes through p_{new} (yellow) and reaches *predicted(x)*.

This algorithm is a deterministic mutation strategy that allows penetrating into as yet uncovered regions of space. Note that this cannot be done perfectly: finding the actual void volume is equivalent to the original visibility problem. However, the adaptive sampling process gives good candidate locations for further sampling rays, namely at discontinuity locations. This strategy works by changing the starting point of the ray instead of its direction.

A discontinuity is detected during the adaptive sampling of an edge by comparing the distance of the ray origin to the actual hitpoint $|hit(x) - x_p|$ with the distance to a "predicted" hitpoint $|predicted(x) - x_p|$. The predicted hitpoint is just the intersection of the ray *x* with the plane of the original triangle *t*. If the new hitpoint is considerably (Δ) closer, i.e.,

$|predicted(x) - x_p| - |hit(x) - x_p| > \Delta$,

the ray is obviously occluded by a closer triangle. Note that we do not check the reverse case (jump from closer to farther triangle) as this will be detected when doing adaptive border sampling for the farther triangle. We calculate a mutated ray from a different view cell position to the predicted hitpoint so that it passes by the occluding triangle. For this, the plane $p = (x_p, hit(x), hit(x_{old}))$ is intersected with the newly found triangle (x_{old} is the previous ray from which x was generated). On the intersecting line, we select a point p_{new} which lies just outside of the new triangle. The mutated ray is now constructed with $x_{new,dir} = predicted(x) - p_{new}$ as direction vector, and $x_{new,p} = intersect(view\ cell, line(p_{new}, predicted(x))$ as origin (see Figure 21). If the new ray is not contained in the ray space Ω (i.e., it does not intersect the view cell), however, it is discarded.

The new ray x_{new} is now treated as independent ray, and the triangle it intersects will be added for adaptive border sampling like any other triangle, but this time with the new view cell origin.

For the 2D example in Figure 21, reverse sampling corresponds to a horizontal movement in ray space.

3.4.3.4. Combining the Different Sampling Algorithms

The sampling strategies presented so far can be combined into an extremely efficient guided visibility sampling algorithm. Its two main components are a sample generator for exploring the ray space with independent random samples, and a sampling queue for propagating the ray using adaptive border sampling and reverse sampling. The algorithm is described by the following pseudocode:

```
main()
 while not finished
  (xp,xdir) = generate_random_ray()
  handle_ray(x)
  while not queue.empty()
   adaptive_border_sampling(
    queue.dequeue())

handle_ray(x)
 if v(x) notin PVS
  PVS += v(x)
  queue += x

adaptive_border_sampling(x)
 t' = enlarge(v(x), ε)
 for each p in t'
  handle_ray((xp, p-xp))
 for each edge (pl, pr) in t'
  subdiv_edge(pl, pr)

subdiv_edge(pl, pr)
 x = (xp, pl-xp)
 y = (xp, pr-xp)
 check_discontinuity(x)
 check_discontinuity(y)

 if v(x) = v(y) or |hit(x)-hit(y)| < ε
  return
 else
  p = (pl + pr)/2
  handle_ray((xp, p-xp))
  subdiv_edge(pl, p)
  subdiv_edge(p, pr)

check_discontinuity(x)
 if |predicted_hit(x) - xp)| -
    |hit(x) - xp| > thresh)
  xnew = reverse_sampling(x)
  if start(xnew) in view cell
   handle_ray(xnew)
```

3.4.3.5. Termination criteria

Depending on the application requirements, there are several options regarding when to stop casting rays for a view cell: a) a fixed criterion, allocating a number of rays or an amount of time for the computation of each view cell, or b) an adaptive criterion, terminating if the number of newly found triangles per a certain number of samples falls below a threshold, or most preferably, c) a combination

of both. A typical example for such a criterion is: stop the iteration when not more than 50 new triangles are found for 1M rays, or when a total of 10M rays has been shot, whichever comes first.

3.4.4. Related Work, Discussion and Applications

A large volume of research has been devoted to visibility problems due to their importance in computer graphics, computer vision, robotics and other fields. This section compares various aspects of the proposed visibility sampling algorithm to a wider class of from-region visibility algorithms. For a general overview, we can recommend excellent surveys of visibility problems and algorithms.

From-region visibility algorithms are usually classified as exact (potentially visible set PVS = exact visible set EVS), conservative ($PVS \supseteq EVS$), aggressive ($PVS \subseteq EVS$), or approximate ($PVS \sim EVS$).

3.4.4.1. Exact Visibility

Exact solutions to compute visibility from a region in space have been rare, but recently, two algorithms have been published and further improved upon that are both exact and work for general scenes. While exact algorithms have been the holy grail of the visibility community for a long time, these two algorithms show that the complexity inherent in the visibility problem may be an obstacle to make exact visibility widely applicable. The high running times and high complexity of implementation are critical, and numerical robustness issues can actually make the solution as approximate as a sampling-based strategy. We believe that sampling-based methods and exact methods complement each other, as they have different strengths and weaknesses.

3.4.4.2. Conservative Visibility

Several authors stress the importance of conservative visibility computations, i.e., never underestimating the visible set. Since this problem is almost as hard as the exact visibility problem, practically all published conservative from-region algorithms simplify the problem by imposing certain restrictions on the scene. Typical restrictions are the limitation to 2.5D visibility, architectural scenes, the restriction to volumetric occluders, or the restriction to larger occluders close to the view cell---this last restriction is implied by the nature of the data structures used to store visibility information. While it can be argued that larger occluders can be synthesized from smaller ones, this is not possible in general. The guarantee to include all visible geometry in the PVS may be important for some applications, but ultimately, sampling-based methods can be much more successful:

1. As opposed to the published conservative algorithms, they do not make any assumptions about the scene, allowing them to handle a much larger variety of scenes.
2. Due to their ease of implementation and robustness, non-conservative algorithms are more practical for commercial products such as computer games, and are already used in this context.
3. Numerical issues often make conservative algorithms non-conservative in practice.

3.4.4.3. Aggressive Visibility

Since visibility is such a fundamental problem, general, robust and practical tools are important to complement the specialized algorithms discussed before. These tools are almost universally based on sampling. The two most popular solutions are to randomly select a large number of rays to sample visibility, or to first sample the boundary of the view cell with points and then sample visibility from each of these points. In the context of view planning for laser range scanners, sampling algorithms exist that store the void surface or the void volume to compute the next best view. A similar algorithm was also used for the generation of textured depth meshes. Another option is to shoot rays from the scene triangles towards the view cell, which leads to oversampling of ray space for most scenes.

Nirenstein and Blake were the first to realize the full potential of sampling for visibility computation. They proposed a new approach which uses graphics hardware for sampling. This algorithm aims to reduce the rendering time by culling even visible triangles as long as this does not result in significant rendering error. This is opposed to our algorithm, which always tries to find the best possible approximation of the exact visible set.

3.4.4.4. Algorithm Analysis

Ray space analysis. In the introduction in Figure 17, we have argued that it is desirable not to sample the ray space regularly. The right image in this figure shows that only an approximately 1D subspace of rays needs to be considered in this simple 2D example. Our new algorithm samples ray space more intelligently: random sampling places initial seed points in ray space to stochastically search for regions in ray space that have not been explored yet. To continue the example for 2D as in the figure, adaptive border sampling corresponds to a vertical expansion in 2D ray space (since the viewpoint remains fixed) which only proceeds into yet unexplored areas. A particular advantage of the adaptive border sampling method is that the sampling rate is adapted to the geometric complexity of the visible surfaces. Reverse sampling, on the other hand, is a movement in the horizontal direction (since the hitpoint remains fixed) in cases where these movements promise to lead to not yet explored regions.

For the full 3D case, it is instructive to study our algorithm in terms of the *visibility complex*. The visibility complex describes a partition of the 4D ray space into 4D regions of rays that hit the same object (note that ray space is strictly 4D because we are only interested in rays starting from the view cell). The 3D boundaries of this partition are called *tangency volume* and consist of rays tangent to scene objects. Samples placed along the object borders therefore correspond to samples near the tangency volume of the object in dual space. Since we keep the viewpoint (2 degrees of freedom) fixed during the deterministic ABS exploration phase, we need to sample a 1D set only. Without ABS, we would ignore the tangency volumes and have to sample the whole 2D subset of ray space defined by the chosen viewpoint.

Reverse sampling, on the other hand, looks for lines tangent to two scene edges. In ray space, these lines are near intersections of two tangency volumes. These intersections are called bitangents and are only 2D. For reverse sampling, the viewpoint is allowed to move along a plane (1D), so in total RS

also samples a 1D set. The combined ABS and RS strategies therefore correspond to explorations of the 4D ray space along those 1D curves that are most likely to reveal new objects. This explains the high efficiency of the GVS algorithm.

Another useful interpretation of the ABS sampling strategy in 3D is based on the visibility map. The visibility map is a structure that contains all visible line segments in a given view. These segments can be characterized mainly as flat and corner (interior edges of a mesh), or shadow (depth discontinuities). The ABS sampling strategy places samples at all edges of the visibility map (without explicitly constructing it). Samples on interior edges of a mesh serve to find connected sets of a mesh (trivially adjacent regions in the visibility complex). Samples at the shadow edges serve to discover depth discontinuities, where objects are partly occluded by other objects. Shadow edges are where the RS sampling strategy is used to refine the sampling (by finding the bitangents in the visibility complex).

Accuracy. The term conservative (or even exact) visibility is actually quite misleading. Most algorithms, though conservative in theory, are not conservative in practice due to numerical robustness problems. This is especially true for algorithms relying on graphics hardware. Furthermore, complex algorithms are prone to implementation problems. Due to the much improved sampling efficiency, the magnitude of error introduced by our algorithm is comparable to that of other error sources. Such errors are usually tolerated for conservative algorithms. Other algorithms that are often used in conjunction with visibility processing, like level-of-detail algorithms or shadow mapping, are an additional source of errors.

Scene complexity. One distinguishing feature of our sampling-based algorithm is that it can handle arbitrary types of scenes with high overall and visual complexity. It does not rely on occluder synthesis, and depends mostly on the size of the visible set, not on the total scene complexity.

3.4.4.5. Limitations and Future Work

Although guided visibility sampling generally finds the major part of the PVS very quickly, the fact that it is stochastic on the one hand and guided by the visibility in the scene on the other hand makes the final accuracy dependent on the structure of the scene. Therefore, we cannot give any hard guarantees for the pixel error of the calculated PVS. Also, the ability to explore connected ray space subsets in the far distance is limited by the numerical precision of the ray direction vector. For ABS, this means that triangles that have a solid angle of less than double precision accuracy when seen from the ray origin will most likely be missed.

The worst case of scene complexity is in scenes that consist of a large set of small disconnected triangles, such as forest scenes or synthetic scenes of random triangles. The visibility of such scenes is so complex that even sampling-based solutions will either have high error or take a long time to compute. Still, it is important to point out that sampling-based algorithms are the only ones that are able to even process these scenes.

In this respect, an avenue of future work is to incorporate geometric LOD into the sampling framework, similar to the vLOD system proposed by Chhugani et al. Geometric LODs could

potentially increase the speed of the ray tracer, and make intersection computations more robust because small triangles in the distance get replaced by larger ones. However, robust geometric LOD is not available for all scenes, and integrating LODs into ray tracers is a current topic of research. Furthermore, the error metric used to create the LODs impacts the accuracy of the visibility algorithm and therefore the usable output resolutions.

3.4.4.6. Applications

One important strength of sampling-based methods is their ease of application. We will discuss a number of application scenarios for our algorithm.

Visibility preprocessing for real-time rendering and games. This is the scenario already described in the overview, and one of the most important applications for GVS. For example, the scenes of current computer games are becoming increasingly general, so that special purpose algorithms (cells and portals, and 2.5D solutions) cannot be used anymore, while exact algorithms are difficult to implement and error-prone. GVS can be used in all stages of game development: During level design, the number of rays can be limited so that a coarse solution can be provided almost instantaneously. For the final production, the PVS can be calculated with high accuracy. It is very important to create a PVS that is as close to the EVS as possible and not dependent on a particular output resolution, since the resolution the application will be run at is not known in advance. In addition, antialiasing methods (supersampling and multisampling) use information from subpixel triangles, so that the virtual resolution is even higher. Note that although scenes in computer games are inherently dynamic, the major part of the scene is still static, so huge gains in rendering speeds can be obtained. Furthermore, GVS works on arbitrary polyhedral view cells, so that the view space can be chosen freely.

Online and networked visibility. As shown in the results, a reasonable approximation to the EVS with low pixel error can be found in a second or less. Therefore, GVS can be used for online visibility culling by running it on a separate processor or over the network, as described in the Instant Visibility system. In this case, transmitting the PVS on a per-object basis will improve results because it suffices for one triangle of an object to be found by GVS in order to classify the whole object as visible. Furthermore, a small modification to GVS makes the algorithm better suitable to progressive evaluation: instead of interleaving ABS and random samples from the beginning, create a certain number (e.g., 1M) of random samples in a startup phase, and only then use those to seed the ABS rays. This will give a better distribution of samples in the initial phase of the algorithm, since ABS systematically “flood fills” the PVS around its seed point, and it takes some time until all image regions have been reached.

Impostor generation. In many scenes, visibility culling is not sufficient to guarantee a high frame rate everywhere in the model. Therefore, image-based methods can be used to replace complex scene parts by so-called impostors. However, since impostors trade rendering speed against memory consumption, it is important to find the exact visible parts of the scene to avoid wasting impostor memory on invisible geometry. GVS is ideally suited for this purpose since it provides accurate per-triangle visibility information, so that only those object parts that are actually visible need to be stored in an impostor.

Visibility as decision basis. Many practical applications require accurate visibility information as part of a decision making process. Examples include visibility analysis in urban planning (does the new skyscraper impact old town?), military applications (line of sight culling, tactical battlefield management, telecommunications (visibility of emitters), robotics and many more. GVS is advantageous for these problems because it is general purpose and does not have any parameters to tweak, and does not depend on any special properties of the scene.

3.4.5. Conclusion

We have presented a visibility sampling algorithm to compute a full 3D visibility solution from a region in space. The proposed algorithm improves the efficiency of previous sampling strategies by over two orders of magnitude, thereby allowing visibility solutions with negligible error to be computed in reasonable time. The proposed algorithm works on arbitrary so-called polygon soups and does not require any memory beyond that used by the ray caster. Due to the new sampling strategies employed in the algorithm, its accuracy is competitive even with exact and conservative approaches, while it is also extremely simple to implement.

We have provided evidence that Guided Visibility Sampling closes an important gap in visibility research. It combines the speed and ease of implementation of sampling-based and special-purpose conservative algorithms with most of the accuracy of exact solutions. Thus, GVS can be used as a general purpose visibility tool.

3.5. ONLINE VISIBILITY CULLING

This module works in runtime to quickly cull occluded objects given a particular viewpoint and a viewing direction. We have designed and implemented an online culling algorithm which efficiently uses occlusion queries supported by the recent graphics hardware. Large regions of similar visibility are culled early by using a hierarchy. To eliminate CPU stalls and GPU starvation, we exploit temporal coherence and schedule the occlusion queries using a priority queue. To reduce unnecessary queries, we pull up and push down visibility information in the hierarchy.

We further provide a prototype implementation of our online culling algorithm for the Ogre3D engine. The integrated algorithm is usable with most Ogre3D features and could be integrated into other scene manager plugins (using different spatial hierarchies) without major obstacles. It is flexible enough so that it's applicable in both a rendering and a query mode.

3.5.1. Hardware Occlusion Queries

Hardware occlusion queries follow a simple pattern: To test visibility of an occludee, we send its bounding volume to the GPU. The volume is rasterized and its fragments are compared to the current contents of the z-buffer. The GPU then returns the number of visible fragments. If there is no visible fragment, the occludee is invisible and it need not be rendered. There are several advantages of hardware occlusion queries:

- Generality of occluders. We can use the original scene geometry as occluders, since the queries use the current contents of the z-buffer.
- Occluder fusion. The occluders are merged in the z-buffer, so the queries automatically account for occluder fusion. Additionally this fusion comes for free since we use the intermediate result of the rendering itself.
- Generality of occludees. We can use complex occludees. Anything that can be rasterized quickly is suitable.
- Exploiting the GPU power. The queries take full advantage of the high fill rates and internal parallelism provided by modern GPUs.
- Simple use. Hardware occlusion queries can be easily integrated into a rendering algorithm. They provide a powerful tool to minimize the implementation effort, especially when compared to CPU-based occlusion culling.

3.5.2. Algorithm Overview

Our method is based on exploiting temporal coherence of visibility classification. In particular, it is centered on the following three ideas:

- We initiate occlusion queries on nodes of the hierarchy where the traversal terminated in the last frame. Thus we avoid queries on all previously visible interior nodes.

- We assume that a previously visible leaf node remains visible and render the associated geometry without waiting for the result of the corresponding occlusion query.

- Issued occlusion queries are stored in a query queue until they are known to be carried out by the GPU. This allows interleaving the queries with the rendering of visible geometry.

The algorithm performs a traversal of the hierarchy that is terminated either at leaf nodes or nodes that are classified as invisible. Let us call such nodes the termination nodes, and interior nodes that have been classified visible the opened nodes. We denote sets of termination and opened nodes in the i-th frame Ti and Oi, respectively. In the i-th frame, we traverse the kD-tree in a front-to-back order, skip all nodes of Oi−1 and apply occlusion queries first on the termination nodes Ti−1. When reaching a termination node, the algorithm proceeds as follows:

- For a previously visible node (this must be a leaf), we issue the occlusion query and store it in the query queue. Then we immediately render the associated geometry without waiting for the result of the query.

- For a previously invisible node, we issue the query and store it in the query queue.

When the query queue is not empty, we check if the result of the oldest query in the queue is already available. If the query result is not available, we continue by recursively processing other nodes of the kD-tree as described above. If the query result is available, we fetch the result and remove the node from the query queue. If the node is visible, we process its children recursively. Otherwise, the whole subtree of the node is invisible and thus it is culled.

In order to propagate changes in visibility upwards in the hierarchy, the visibility classification is pulled up according to the following rule: An interior node is invisible only if all its children have been classified invisible. Otherwise, it remains visible and thus opened. An example of the behavior of the method on a small kD-tree for two subsequent frames is depicted Figure 22. The pseudo-code of the complete algorithm is given in Figure 4.

The sets of opened nodes and termination nodes need not be maintained explicitly. Instead, these sets can be easily identified by associating with each node an information about its visibility and an id of the last frame when it was visited. The node is an opened node if it is an interior visible node that was visited in the last frame (line 23 in the pseudocode). Note that in the actual implementation of the pull up we can set all visited nodes to invisible by default and then pull up any changes from invisible to visible (lines 25 and line 12). This modification eliminates checking children for invisibility during the pull up.

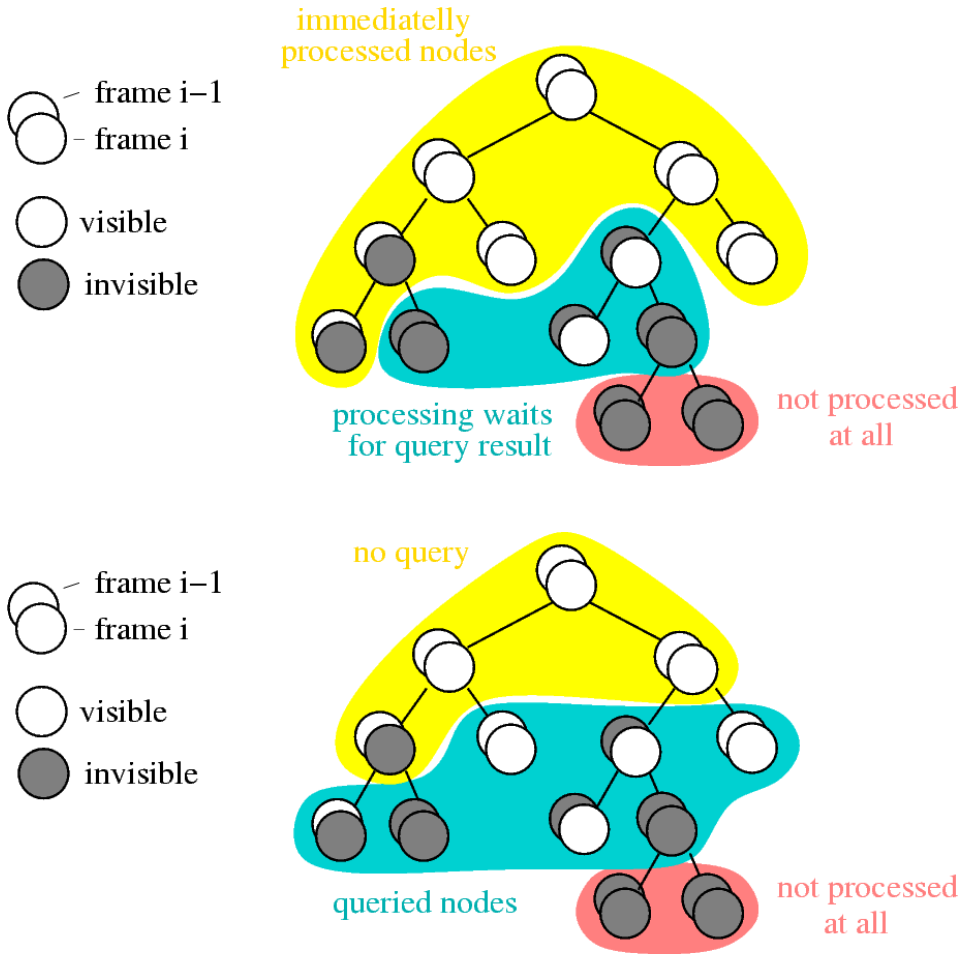

Figure 22, (top) Illustration of processing requirements of nodes of the spatial hierarchy at frame i. (bottom) Illustration of query requirements of nodes of the spatial hierarchy.

3.5.3. Reduction of the number of queries

Our method reduces the number of visibility queries in two ways: Firstly, as other hierarchical culling methods we consider only a subtree of the whole hierarchy (opened nodes + termination nodes). Secondly, by avoiding queries on opened nodes we eliminate part of the overhead of identification of this subtree. These reductions reflect the following coherence properties of scene visibility:

- Spatial coherence. The invisible termination nodes approximate the occluded part of the scene with the smallest number of nodes with respect to the given hierarchy, i.e., each invisible termination node has a visible parent. This induces an adaptive spatial subdivision that reflects spatial coherence of visibility, more precisely the coherence of occluded regions. The adaptive nature of the subdivision allows to minimize the number of subsequent occlusion queries by applying the queries on the largest spatial regions that are expected to remain occluded.

- Temporal coherence. If visibility remains constant the set of termination nodes needs no adaptation. If an occluded node becomes visible we recursively process its children (pull-down). If a visible node becomes occluded we propagate the change higher in the hierarchy (pull-up). A pull-down reflects a spatial growing of visible regions. Similarly, a pull-up reflects a spatial growing of occluded regions. By avoiding queries on the opened nodes, we can save 1/k of the queries for a hierarchy with branching factor k (assuming visibility remains constant). Thus for the kD-tree, up to half of the queries can be saved. The actual savings in the total query time are even larger: the higher we are at the hierarchy, the larger boxes we would have to check for occlusion. Consequently, the higher is the fill rate that would have been required to rasterize the boxes. In particular, assuming that the sum of the screen space projected area for nodes at each level of the kD-tree is equal and the opened nodes form a complete binary subtree of depth d, the fill rate is reduced (d + 2) times.

3.5.4. The Reduction of CPU stalls and GPU starvation

The reduction of CPU stalls and GPU starvation is achieved by interleaving occlusion queries with the rendering of visible geometry. The immediate rendering of previously visible termination nodes and the subsequent issuing of occlusion queries eliminates the requirement of waiting for the query result during the processing of the initial depth layers containing previously visible nodes. In an optimal case, new query results become available in between and thus we completely eliminate CPU stalls. In a static scenario, we achieve exactly the same visibility classification as the hierarchical stop-and-wait method.

If the visibility is changing, the situation can be different: if the results of the queries arrive too late, it is possible that we initiated an occlusion query on a previously occluded node A that is in fact occluded by another previously occluded node B that became visible. If B is still in the query queue, we do not capture a possible occlusion of A by B since the geometry associated with B has not yet been rendered. We show that the increase of the number of rendered objects compared to the stop-and-wait method is usually very small.

3.5.5. Further Optimizations

This section discusses a couple of optimizations of our method that can further improve the overall rendering performance. In contrast to the basic algorithm from the previous section, these optimizations rely on some user specified parameters that should be tuned for a particular scene and hardware configuration.

Conservative visibility testing

The first optimization addresses the reduction of the number of visibility tests at the cost of a possible increase in the number of rendered objects. This optimization is based on the idea of skipping some occlusion tests of visible nodes. We assume that whenever a node becomes visible, it remains visible for a number of frames. Within the given number of frames we avoid issuing occlusion queries and simply assume the node remains visible. This technique can significantly reduce the number of visibility tests applied on visible nodes of the hierarchy. Especially in the case of sparsely occluded scenes, there is a large number of visible nodes being tested, which does not provide any benefit since most of them remain visible. On the other hand, we do not immediately capture all changes from

visibility to invisibility, and thus we may render objects that have already become invisible from the moment when the last occlusion test was issued.

In the simplest case, the number of frames a node is assumed visible can be a predefined constant. In a more complicated scenario this number should be influenced by the history of the success of occlusion queries and/or the current speed of camera movement.

Approximate visibility

The algorithm as presented computes a conservative visibility classification with respect to the resolution of the z-buffer. We can easily modify the algorithm to cull nodes more aggressively in cases when a small part of the node is visible. We compare the number of visible pixels returned by the occlusion query with a user specified constant and cull the node if this number drops below this constant.

Complete elimination of CPU stalls

The basic algorithm eliminates CPU stalls unless the traversal stack is empty. If there is no node to traverse in the traversal stack and the result of the oldest query in the query queue is still not available, it stalls the CPU by waiting for the query result. To completely eliminate the CPU stalls, we can speculatively render some nodes with undecided visibility. In particular, we select a node from the query queue and render the geometry associated with the node (or the whole subtree if it is an interior node). The node is marked as rendered but the associated occlusion query is kept in the queue to fetch its result later. If we are unlucky and the node remains invisible, the effort of rendering the node's geometry is wasted. On the other hand, if the node has become visible, we have used the time slot before the next query arrives in an optimal manner.

To avoid the problem of spending more time on rendering invisible nodes than would be spent by waiting for the result of the query, we select a node with the lowest estimated rendering cost and compare this cost with a user specified constant. If the cost is larger than the constant we conclude that it is too risky to render the node and wait till the result of the query becomes available.

Initial depth pass

To achieve maximal performance on modern GPUs, one has to take care of a number of issues. First, it is very important to reduce material switching. Thus modern rendering engines sort the objects (or patches) by materials in order to eliminate the material switching as good as possible. Next, materials can be very costly, sometimes complicated shaders have to be evaluated several times per batch. Hence it should be avoided to render the full material for fragments which eventually turn out to be occluded. This can be achieved by rendering an initial depth pass (i.e., enabling only depth write to fill the depth buffer). Afterwards the geometry is rendered again, this time with full shading. Because the depth buffer is already established, invisible fragments will be discarded before any shading is done. This approach can be naturally adapted for use with the CHC algorithm. Only an initial depth pass is rendered in front-to-back order using the CHC algorithm. The initial pass is sufficient to fill the depth buffer and determine the visible geometry. Then only the visible geometry is rendered again, exploiting the full optimization and material sorting capability of the rendering engine. If the materials require several rendering passes, we can use a variant of the depth pass method. We render only the first passes using the algorithm (e.g., the solid passes), determining the visibility of the patches, and render all the other passes afterwards. This approach can be used when there are passes which require

a special kind of sorting to be rendered correctly (e.g., transparent passes, shadow passes). In Figure 23, we can see that artifacts occur in the left image if the transparent passes are not rendered in the correct order after applying the hierarchical algorithm (right image). In a similar fashion, we are able to handle shadows as shown in Figure 24.

Batching multiple queries

When occlusion queries are rendered interleaved with geometry, there is always a state change involved. To reduce state changes, it is beneficial not to execute one query at a time, but multiple queries at once. Instead of immediately executing a query for a node when we fetch it from the traversal stack, we add it to the pending queue. If n of these queries are accumulated in the queue, we can execute them at once. To obtain an optimal value for n, several some heuristics can be applied, e.g., a fraction of the number of queries issued in the last frame.

Figure 23, (left) all passes are rendered with CHC. Note that the soldiers are visible through the tree. (right) Only the solid passes are rendered using CHC, afterwards the transparent passes.

Figure 24, The online culling module correctly handles shadow volumes.