Advanced Tools for Developing Highly Realistic Computer Games

Finished Modules for Illumination

Document identifier:	**GameTools-5-D5.4-03-1-1-Finished Illumination Modules**
Date:	**03/04/2007**
Work package:	**WP5: Illumination**
Partner(s):	**BUTE, UdG, Unilim**
Leading Partner:	**BUTE**
Document status:	**Approved**
Deliverable identifier:	**D5.4**

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

Delivery Slip

	Name	Partner	Date	Signature
From	Laszlo Szirmay-Kalos	BUTE	10.03.2007	
Reviewed by	Moderator and reviewers	ALL		
Approved by	Moderator and reviewers	ALL		

Document Log

Issue	Date	Comment	Author
1-0	03.03.2007	First draft	Laszlo Szirmay-Kalos
1-1	08.03.2007	Final version	Laszlo Szirmay-Kalos

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files / URL
Word	GameTools-5-D5.4-03-1-1-Finished Illumination Modules.doc

CONTENT

1. INTRODUCTION 4

1.1. OBJECTIVES OF THIS DOCUMENT 4
1.2. DOCUMENT AMENDMENT PROCEDURE 4
1.3. TERMINOLOGY 4

2. STRUCTURE OF THE ILLUMINATION WORKPACKAGE 6

3. DESCRIPTION OF THE RUN-TIME ILLUMINATION MODULES 7

3.1. OGRE3D VERSION ILLUMINATION MODULES 8
3.1.1. Extending Ogre3D Material Scripts 8
3.1.2. Compiling the Ogre3D version of the illumination modules 9
3.1.3. Game requirements 10
3.1.4. Implemented techniques of the Ogre3D version 13
3.1.5. Building modules with techniques in Ogre3D 28
3.2. SHARK3D VERSION OF ILLUMINATION MODULES 51
3.2.1. Implemented shader components of the Shark3D version 51
3.2.2. GPU programs in Shark3D 60
3.2.3. Building modules with shader components in Shark3D 62

4. FILE FORMATS 73

4.1. COLLADA FILE [MEDIA*.DAE] 73
4.2. LEVEL FILE [MEDIA*.LEVEL] 73
4.3. MATERIAL FILE [MEDIA*.MATERIALS 74
4.4. MESH FILES [MEDIA*.MESH.XML, PROCESSEDMESHES*.MESH.XML, *.MESH] 74
4.5. ENTRY POINTS FILE [PRM\PRMENTRYPOINTS.TEXT] 74
4.6. PRM TEXTURES [PRM*DDS] 74

5. MAYA SCENE EXPORTER 75

6. LIGHT PATH MAPS PREPROCESSOR 78

6.1. RUNNING THE PATH MAP PREPROCESSOR 79
6.2. CONTROLS USED IN PATH MAP PREVIEW RENDERING 80
6.3. PATH MAP ATTRIBUTES IN MAYA 81

7. OBSCURANCES PREPROCESSOR 85

7.1. IMPLEMENTATION OF THE OBSCURANCES PREPROCESSOR 86
7.2. RUNNING THE OBSCURANCES PREPROCESSOR 87

8. BILLBOARD TREE PREPROCESSOR 90

8.1. RUNNING THE BILLBOARD TREE GENERATOR 91
8.2. BILLBOARD TREE PLUGIN IN MAYA 94
8.3. MODELLING BILLBOARD CLOUD TREES IN 3DS MAX 95

1. INTRODUCTION

1.1. OBJECTIVES OF THIS DOCUMENT

This document describes the finished, integrated modules for the Illumination Work Package. Its aim is to describe the modules and explain how they work.

1.2. DOCUMENT AMENDMENT PROCEDURE

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

1.3. TERMINOLOGY

Glossary

component	An equivalent to *technique* according to the Shark3D terminology.
final rendering	A *pass* or series of passes rendering into the frame buffer and thus generating the image. All other passes render into the texture memory.
GPU	Graphics Processing Units
GTP	Game Tools Project
GUI	Graphical User Interface
effect	An illumination phenomenon, such as mirror like reflection, refraction, metallic reflection, diffuse inter-reflection, soft shadow, caustics, light scattering in participating medium, etc.
illumination manager	The runtime operational framework for illumination modules. It manages instantiations, invocations, and dependencies of techniques.
level	A file that describes the set of objects and their illumination properties in a scene.
material	A definition of rendering properties.
module	A complete solution to generate a particular illumination *effect.*
pass	A rendering cycle when the geometry is sent through the rendering pipeline once.
preprocessor	A stand-alone program or a Maya plugin that prepares geometry or texture to be used in the on-line rendering program. The preprocessor should be run only once during the preparation of the model.
PRM or Light Path Map	Precomputed radiance map used in the light path map tool.
RenderTechnique	Base class for rendering techniques in the GTP illumination modules.

resource	Data generated by a technique in the GTP illumination modules, usually in the form of a texture, used as input in further techniques or in the *final rendering*.
shader	A program written in Cg or HLSL and running on the GPU, either on its vertex or on its fragment processor. Also, a shader in Shark3D terminology is the equivalent of a *material*, but this is only used in the context of Shark3D.
svn	The GTP repository.
technique	An implementation building block of tools. A technique may be shared by different *tools*. A technique is a way of rendering either into the frame buffer or into the texture memory. A technique may have a vertex and a fragment *shader*.
tool	A complete solution to generate a particular illumination *effect*. It is also called *module*, or to emphasize its role, as *GTP illumination module*.
WP	Work Package
/gametools	The root of the gametools repository at www.gametools.org

2. STRUCTURE OF THE ILLUMINATION WORKPACKAGE

The illumination work package is a collection of solutions called *modules* or *tools* to render scenes realistically, providing physically plausible global illumination effects. The tools themselves are responsible for different lighting phenomena, including *diffuse/glossy inter-reflections, mirror reflections, refractions, caustics, single and multiple light scattering in participating medium, obscurances, shadows,* etc. When the illumination work package was delivered as a collection of stand-alone applications, each tool corresponded to a different program. Each tool had a program part running on the CPU, had different shaders running on the GPU, and used separate resources like textures or cube maps. Some tools like the *billboard cloud tree* or the *light path map* required resources like textures and geometry prepared off line. It means that some tools also had *preprocessors* responsible for these off line computations.

During the integration of GTP illumination modules into game engines like Ogre3D or Shark3D, the functionality of the modules had to be re-organized in order to allow the reuse common functionalities in different tools and also to meet the particular requirements of these engines. While shaders remained the same and preprocessors were changed only with respect to the input requirements of the game engines, CPU programs organizing these shaders had to be significantly restructured. The goal of the restructuring is to eliminate redundancy and to allow shaders to work together via passing information or sharing resources.

The central feature of this framework is the *technique*. A technique is a way to render something: it typically includes shaders and results rendered by these shaders into textures. Instances of techniques are mostly linked to objects, but techniques creating more general resources might be global or linked to light sources. These technique instances are created automatically based on materials assigned to objects, or from the program code. All the different illumination effects generated by the work package tools need some kind of resources during the final rendering step. These resources are usually textures. To create them, one or more rendering passes are necessary. These pre-computing passes are implemented as techniques, and corresponding final rendering shaders use the results. Thus, technique instances assigned to rendered objects will define what resources they require, and these resources might be provided by other techniques.

With the emergence of the concept of techniques, the module or tool ceased to exist as an implementation building block unless the module has a single technique. However, modules are there conceptually, as solutions delivering an illumination effect. Section 3.1.4.12 describes how modules are built from techniques. To clarify the concepts of techniques and modules, we can take an example. The shaders and the functionality of generating a cube map storing distance values in its texels were needed both in the diffuse inter-reflection tool and in the mirror like reflection tool. During integration, we have identified the cube map generation as a specific technique that is used by both solutions. Currently the diffuse inter-reflection tool is set of connection rules that defines how the available techniques should be tied together to produce the desired effect.

During integration the functionality of preprocessors remained the same for the integrated version, only the specific input requirements of Ogre3D and Shark3D had to be taken into consideration.

In the following chapters we first review the general aspects of the integration of the techniques into the game engines, then particular techniques are discussed. This is the run-time part of the illumination modules. The examples show how different techniques should be connected together to obtain the tools responsible for different effects. Finally, we present the preprocessors one by one.

3. DESCRIPTION OF THE RUN-TIME ILLUMINATION MODULES

We have implemented two sets of integrated run-time illumination modules, one for the Ogre3D engine and one for the Shark3D engine, using the same set of shaders. The modules are built of techniques that are shared by different modules.

The following rendering process is formulated according to Ogre3D terminology, but the concept applies to both engines. In Shark3D, a *material* would be referred to as a *shader*, a *technique* as a *shader component*, and shared resources are more limited. The basic mechanism of rendering a visible object proceeds as follows:

- Based on the definition of its material, an object will have a final rendering shader, and associations to render techniques that provide input to this shader.
- If an object is visible, it will trigger the execution of techniques providing necessary resources. Whenever the resource is already available, the technique needs not to be invoked again. Therefore, if more objects trigger the same technique, it will only run once.
- Global and light source techniques are invoked to produce global resources like indirect illumination weights, screen depth information of shadow maps. The techniques are only performed if there is an object in the scene that requires their output resources.
- Technique instances linked to objects are triggered in a similar manner. For instance, a visible caustic receiver object triggers the execution of the caustic caster object's techniques rendering the caustic cube map.
- Most often, objects trigger techniques associated with themselves. For instance, an object featuring localized reflections will require a distance cube map. For better performance, however, multiple objects may share such resources.
- Some resources cannot be simply used as an input to an existing shader pass, but require their own shader pass to be added to the final rendering of the object wishing to make use of the resource. For instance, an object using PRM indirect illumination will add indirect illumination in an extra pass. In such cases, the new pass will be added to the final rendering material of the object.
- After all necessary techniques have been run, and all the resources are available and set as inputs for the shaders, the engine performs rendering. This will invoke shaders aware of and able to use the resources produced by the illumination module techniques.

Two different techniques may require the same resources (e.g.: a distance impostor cube map), so these resources can be shared between them. It is also possible that two objects that use two, possibly different techniques, can have common resources. This can happen in case of global resources (e.g.: scene depth map) or per light resources (e.g.: depth shadow map from a light source), but it can also happen in more general cases (e.g.: for efficiency reasons we would like to compute a common cube map for several small objects that are close to each others). Managing shared resources in case of moving objects is complicated, as resources need to be joined and split dynamically.

The *illumination manager* automates the creation of resources, the assignment of resources to the different instances and the assignment of instances to be used to the objects. Since these organization

tasks depend on the engine, the illumination manager implementation is engine specific, but the shaders of the techniques are similar.

3.1. OGRE3D VERSION ILLUMINATION MODULES

In order to incorporate the GTP illumination modules into the Ogre3D game engine, we extended the material script system and added header files and a library of the new classes. For a game to use the modules, the library of the GTP illumination modules has to be compiled, and invoked from the game according to the declarations. The functionality of the illumination manager is provided by the *OgreIllluminationManager* class. This is the primary interface for setting up illumination techniques.

3.1.1. Extending Ogre3D Material Scripts

The Ogre3D version of the GTP illumination modules has been designed to fit to the concepts of a typical Ogre3D application. Thus, the Ogre3D material framework is used to define how an object is rendered. Ogre3D material scripts have been extended to define illumination techniques.

The class representing an illumination technique is the *RenderTechnique*. The illumination manager must instantiate descendant classes, which implement specific techniques. What technique instances must be constructed depends on how the scene objects are to be rendered. Final rendering will be performed by the engine in the regular manner, performing passes referenced in materials, which materials have been constructed from material scripts. These final rendering passes use vertex and pixel shaders, the inputs of which have to be generated. This will be the task of techniques. Therefore, if a material script defines a pass that uses a shader which requires some input from a technique, this information should be given in the material script itself. This way, the illumination manager can automatically construct the techniques required for a final rendering pass, and assign its results to the final rendering shaders.

In programming terms, *RenderTechniques* should operate on demand of a *Pass* in an object's material. Most techniques set the resources they output as some input of a *Material*, typically a rendered texture to a texture unit state of a *Pass*. It is just natural to define *RenderTechniques* to be used in the material script within the pass that requires the technique. Thus we added the new keyword ***IllumTechniques*** within the pass definition of materials. Within this scope we can define the techniques to be used for the given pass with the keyword ***RenderTechnique***. This keyword needs a parameter that defines the type (referred by *Name* in the **Technique Reference** (section 3.1.4)) of the technique. Within the *RenderTechnique* scope we can set some technique specific parameters, which will be passed to the technique constructor. An example of this technique definition which defines an object as a caustic and a shadow receiver is the following:

```
material exampleMaterial
{
    technique
    {
       pass   // this pass will use Illumination Module techniques
       {
           IllumTechniques // this scope will define techniques to use
           {
               RenderTechnique DepthShadowReceiver // defines a shadow receiver technique
               {
                   max_light_count 1  // attribute of depth shadow receiver technique
               }
               RenderTechnique CausticReceiver // defines a caustic receiver technique
               {
                   max_caster_count 2  // attribute of caustic receiver technique
```

```
            }
        }
        // … usual pass definition
    }
  }
}
```

Another example that shows how a technique can output its resource to a specified input of the pass (renders and binds a cube texture to a specific texture unit of the pass) is as follows:

```
material exampleMaterial2
{
    technique
    {
       pass   // this pass will use Illumination Module techniques
       {
           IllumTechniques // this scope will define techniques to use
           {
               RenderTechnique ColorCubeMap // this scope will define a color cube map
               {
                   resolution 256  // resolution of the cubemap to be created
                   texture_unit_id 0  // the texture unit this cubemap in this pass
               }
            }
            // … pass attributes like gpu program references etc.
            texture_unit // the cubemap is this is the first texture unit (id = 0)
            {
            }
            // .... other texture units if needed
       }
    }
}
```

For more information about implemented techniques and their parameters see section 3.1.4.

3.1.2. Compiling the Ogre3D version of the illumination modules

All Ogre3D projects that incorporate the GTP illumination modules have to use a library called IllumModule (this is an abstract interface) and its Ogre3D implementation called OgreIllumModule. The source code for these libraries can be found in the GTP repository in the following path:

/gametools/gtp/trunk/Lib/Illum/IllumModule

There is a prepared Visual Studio 7.1 solution file that can be used to compile these libraries:

/gametools/gtp/trunk/Lib/Illum/shared/scripts/GTPIllumination.7.1.sln

This solution also contains the projects of Ogre3D demo applications that demonstrate the integrated modules. The source code of these demos can be found in the repository in:

/gametools/gtp/trunk/App/Demos/Illum/Ogre/src

The OgreIllumModule (and also the demo applications) use Ogre3D 1.2 with the Ogre3D changes that can be found in the *svn* repository:

/gametools/ogre/trunk/ogre_changes/Ogre1.2

These Ogre3D changes should be copied to the Ogre3D 1.2 directory and **Ogre3D should be rebuilt**.

It is important to set the preprocessor definition **GAMETOOLS_ILLUMINATION_MODULE** for all the Ogre3D projects. It is also important to add **SpriteSet.h, SpriteSet.cpp, SpriteParticleRenderer.h,** and **SpriteParticleRenderer.cpp** to the OgreMain project. To compile the OgreIllumModule, an environment variable named **OGRE_PATH** should also be set to the Ogre3D root directory. After these steps the build of the libraries and demos should succeed without errors.

After the compile process the following library files will be created:

/gametools/gtp/trunk/Lib/Illum/IllumModule/IllumModule/bin/release/IllumModule.lib
/gametools/gtp/trunk/Lib/Illum/IllumModule/IllumModule/bin/debug/IllumModule.lib
/gametools/gtp/trunk/Lib/Illum/IllumModule/OgreIllumModule/bin/release/IllumModule_Ogre.lib
/gametools/gtp/trunk/Lib/Illum/IllumModule/OgreIllumModule/bin/debug/IllumModule_Ogre.lib

3.1.3. Game requirements

To use the Ogre3D version of the GTP illumination modules in an Ogre3D game, the following steps are required:

Project Configuration Changes

- Define C preprocessor definition GAMETOOLS_ILLUMINATION_MODULE with the #define directive.
- Add additional library dependencies: IllumModule.lib and OgreIllumModule.lib (also add additional library directories according to the exact locations of these library files, taking care of the current configuration type: debug or release).

Source code changes

In the followings we assume that the Ogre3D application is based on the examples provided with Ogre3D, namely the main class is derived from *ExampleApplication.* Other implementations should change their code not just as it is stated here, but implicitly according the structure of their application.

- Include ***"OgreIlluminationManager.h"*** and also add the additional include directory where this file is located:

  /gametools/gtp/trunk/Lib/Illum/IllumModule/OgreIllumModule/include

- After adding all the objects to the scene graph (e.g.: in the last line of *createScene()*) call

  OgreIlluminationManager::getSingleton().initTechniques();

  OgreIlluminationManager::getSingleton().initTechniques() initializes the techniques that were given by the material scripts. This function searches the scene graph. For each object that has a material with an illumination technique defined, it creates these techniques. For objects that are added to the scene graph later, the following explicitly call can be used:

  OgreIlluminationManager::getSingleton().initTechniques(Entity e),*

  to initialize the technique only for this entity.

- Add the Ogre3D Illumination Manager Singleton as a frame listener in createFrameListener() with this code line:

  mRoot->addFrameListener(&OgreIlluminationManager::getSingleton());

  This call is needed because illumination techniques require resources like textures etc. These resources should be updated in each frame before rendering to the screen. As the *OgreIlluminationManager* singleton is also a *FrameListener,* it will be called in each frame, if it is added to the active frame listeners. One thing that should be considered here is the priority of the frame listeners. *OgreIlluminationManager* should be called after all the other frame listeners, because frame listeners can change object positions and *OgreIlluminationManager* executes rendering processes that require data consistent with the data used when rendering to the frame buffer. This can be ensured with the use of the *FrameListener::setPriority(int)* method (note that this is part of the Ogre3D changes made by GTP Illumination Work Package). The priority of *OgreIlluminationManager* should be set higher than the priority of other frame listeners to be executed later (the default priority of frame listeners is 1, except for *ControllerManager::mFrameTimeController* that has 0 priority).

3.1.3.1. Additional parameters of OgreIlluminationManager

The illumination manager also needs some information that is not provided in the material scripts. These can be given explicitly by code and cover the following topics:

- ***Basic information about the main camera (required):***
  - *OgreIlluminationManager::setMainCamera(Camera *camera)* passes a pointer to the avatar's camera.
  - *OgreIlluminationManager::setMainViewport(Viewport *viewport)* passes a pointer to the main viewport.

- ***Resource joining:***

  Resource joining means that if two objects are close enough to be treated as a single object and they need the same resource (e.g. both needs the color cube map of the surrounding environment), then they can use a single shared resource. This saves us processing time but reduces quality and visual richness (e.g. if two reflective objects share the same environment map, they do not show up in the reflection on the other object). The metric used to decide whether two (or more) objects can be joined is the radius of their minimum enclosing sphere. If it is below a given threshold value, the resources can be shared. The value should depend on the size of the scene. The bounding radius can be given for all resource types or for a specific type.
  - *OgreIlluminationManager::setMaxJoinRadius(float rad)* sets the maximum bounding sphere radius for all resource types.
  - *OgreIlluminationManager::setMaxJoinRadius(RenderingRunType type, float rad)* sets the maximum bounding sphere radius for a specific resource type.

- ***Shadow mapping:***

  The GTP illumination modules implement shadow mapping independently of Ogre3D's built in shadowing techniques. If an object should receive shadows computed by the GTP illumination module, its material should refer to the *DepthShadowReceiver* technique. This will create additional shadow rendering passes to the material. The properties of this shadow mapping can be set with the following functions:

  - *setShadowMapSize(unsigned int size)* sets the size of the shadow maps to use.
  - *setShadowMapMaterialName(String name)* sets the name of the materials that should be used while rendering the shadow maps.
  - *setFocusingSM(bool use)* sets whether or not the light projection should be focused.
  - *setFocusingMapSize(unsigned int size)* specifies the size of the texture that is used in focusing (higher resolution yields more precise focusing but it is slower).
  - *setUseLISPSM(bool use)* sets whether or not *Light Space Perspective* correction should be used (implemented only for directional lights).
  - *setBlurShadowMap(bool use)* sets if the shadow map should be blurred (useful if *Variance Shadow Mapping* is used).

3.1.3.2. Example application

As a practical example of implementing the previously described game requirements, and as a starting point of new applications based on GTP illumination modules, the example application bundled with Ogre3D has to be augmented as follows. Thereafter, any object loaded into the scene graph with materials referencing illumination techniques will be rendered appropriately.

```
class App : public ExampleApplication
{
    protected:

    void createScene(void)
    {
        //usual Ogre code (mesh loading, transformations, material setup)
        ....

        //set global parameters
           //required parameters for main camera and viewport identification
           OgreIlluminationManager::getSingleton().setMainCamera(mCamera);
           OgreIlluminationManager::getSingleton().setMainViewport(mWindow->getViewport(0));
           //resource joining parameters
           //set boundig sphere radius for all run type
           OgreIlluminationManager::getSingleton().setMaxJoinRadius(400);
           //set boundig sphere radius for a given run type
           OgreIlluminationManager::getSingleton().setMaxJoinRadius(ILLUMRUN_PHOTONMAP, 200);
           //shadow mapping attributes (attributes for spotlight using variance shadow)
           OgreIlluminationManager::getSingleton().setShadowMapSize(512);
           OgreIlluminationManager::getSingleton().setFocusingSM(false);
           OgreIlluminationManager::getSingleton().setUseLISPSM(false);
           OgreIlluminationManager::getSingleton().setBlurShadowMap(true);
           OgreIlluminationManager::getSingleton().
               setShadowMapMaterialName("GTP/Basic/Distance_Normalized");
        //search for techniques and initialize them
        OgreIlluminationManager::getSingleton().initTechniques();
```

```
    }

    void createFrameListener(void)
    {
        // This is where we instantiate our own frame listener
        mFrameListener= new myFrameListener(...);
        // Set priority of the frame listener.
        // The default priority value is 1
        mFrameListener->setPriority(2);
        // set priority of OgreIlluminationManager.
        // It should be higher that the other frame listener's
        OgreIlluminationManager::getSingleton().setPriority(3);
        // add the frame listeners to the active frame listeners
        mRoot->addFrameListener(mFrameListener);
        mRoot->addFrameListener(&OgreIlluminationManager::getSingleton());
    }
}
```

3.1.4. Implemented techniques of the Ogre3D version

This section describes the rendering techniques that have been implemented in the Ogre3D version of the GTP illumination modules. These techniques will be shared by the module.

The final gathering module responsible for ideal reflections and refractions with localized environment mapping, metallic materials, and glow is implemented in the following techniques:

- ColorCubeMap
- DistanceCubeMap

Techniques related to caustic generation are:

- CausticCaster
- CausticReceiver

Diffuse inter-reflections with pre-convolved environment mapping are provided by the following technique:

- ReducedColorCubeMap

The light path map algorithm is supported by the following run-time technique:

- PathMap

Techniques related to particle systems and volumetric media

- SphericalBillboard
- Fire
- HPS (hierarchical particle systems)
- IllumVolume

Realistic soft shadows are implemented in technique called

- DepthShadowReceiver

The illumination manager collects and organizes these techniques, standardizes them, manages the creation and update of render targets, and sets the appropriate shaders for rendering. In the following subsections, we consider every technique in detail.

3.1.4.1. Color cube map render technique

This technique generates a color cube map. The color cube map can be used to display reflections and refractions.

Name
ColorCubeMap

Parameters

Name	Type	Default Value	Description
update_interval	unsigned int	1	update frequency (if 0 only updates once)
Start_frame	unsigned int	1	adds an offset to the current frame number to help evenly distribute updates between frames (the frame number of the first update)
resolution	unsigned int	256	cube map resolution
texture_unit_id	unsigned int	0	the id of the texture unit state the resulting cube map should be bound to
distance_calc	bool, float	true 2.0	flag to skip cube face update if object is far away - bool value to enable calculation - float value defines the distance tolerance used in face skip
Face_angle_calc	bool, float	true 2.0	flag to skip cube face update if face is negligible - bool value to enable calculation - float value defines angle tolerance used in face skip
update_all_face	bool	False	defines if all cube map faces should be updated in a frame or only one face per frame
render_self	bool	False	sets if the object should be rendered to the cube map
render_env	bool	True	sets if the environment should be rendered to the cube map

self_material	string	""	the material that should be set for the object while rendering the cube map (if no string is given the original materials are used)
env_material	string	""	the material that should be set for the environment while rendering the cube map (if no string is given the original materials are used)
attach_to_texture_unit	bool	True	sets if this cube map should be attached to a texture unit of the pass
layer	unsigned int	0	the layer of this cube map (multiple layers are used by shaders dealing with multiple reflections and refractions)
get_min_max	bool	0	sets if the minimum and maximum values of the cube map should be computed (it is quite a slow process as it requires texture read back to system memory - it is used to speed up linear search in cube maps while finding intersection points in multiple reflections)
min_var_name	string	""	sets the name of the GPU fragment shader program parameter to which the minimum value should be bound to (if no string is given the variable is not bound)
max_var_name	string	""	sets the name of the GPU fragment shader program parameter to which the maximum value should be bound to (if no string is given the variable is not bound)

3.1.4.2. Distance cube map render technique

This technique generates a distance impostor cube map (a cube map of the distance of the surrounding environment from the cube map center). The distance impostor is a sampled representation of the scene and makes it possible to find intersection points of an arbitrary ray with the environment which makes reflection and refraction calculations more accurate.

Name
DistanceCubeMap

Parameters

Name	Type	Default Value	Description
update_interval	unsigned int	1	update frequency (if 0 only updates once)

Start_frame	unsigned int	1	adds an offset to the current frame number to help evenly distribute updates between frames (the frame number of the first update)
resolution	unsigned int	256	cube map resolution
texture_unit_id	unsigned int	1	the id of the texture unit state the resulting cube map should be bound to
distance_calc	bool, float	true 2.0	flag to skip cube face update if object is far away - bool value to enable calculation - float value defines the distance tolerance used in face skip
Face_angle_calc	bool, float	true 2.0	flag to skip cube face update if face is negligible - bool value to enable calculation - float value defines angle tolerance used in face skip
update_all_face	bool	false	defines if all cube map faces should be updated in a frame or only one face per frame
render_self	bool	false	sets if the object should be rendered to the cube map
render_env	bool	true	sets if the environment should be rendered to the cube map
self_material	string	"GTP/Basic/ Distance"	the material that should be set for the object while rendering the cube map (if no string is given the original materials are used)
env_material	string	"GTP/Basic/ Distance"	the material that should be set for the environment while rendering the cube map (if no string is given the original materials are used)
attach_to_texture_unit	bool	true	sets if this cube map should be attached to a texture unit of the pass
layer	unsigned int	0	the layer of this cube map (multiple layers are used by shaders dealing with multiple reflections and refractions)
get_min_max	bool	0	sets if the minimum and maximum values of the cube map should be computed (it is quite a slow process as it requires texture read back to system memory - it is used to speed up linear search in cube maps while finding intersection points in multiple reflections)

Min_var_name	string	""	sets the name of the GPU fragment shader program parameter to which the minimum value should be bound to (if no string is given the variable is not bound)
max_var_name	string	""	sets the name of the GPU fragment shader program parameter to which the maximum value should be bound to (if no string is given the variable is not bound)

3.1.4.3. Caustic caster technique

This technique specifies that the object can cast caustics.

Name
CausticCaster

Parameters

Name	Type	Default Value	Description
update_interval	unsigned int	1	update frequency (if 0 only updates once)
start_frame	unsigned int	1	adds an offset to the current frame number to help evenly distribute updates between frames (the frame number of the first update)
photonmap_resolution	unsigned int	64	photon map resolution
caustic_cubemap_resolution	unsigned int	128	caustic cube map resolution
photon_map_material	string	"GTP/Caustic/PhotonMap_HitEnv"	the name of the material should be used when rendering the photon hit map - the default material needs a distance impostor cube map and finds photon hits with secant search
caustic_map_material	string	"GTP/Caustic/CauCube_PointSprite"	the name of the material that should be used when rendering the caustic cube map
photon_map_tex_id	unsigned int	0	the texture unit state id of the caustic map generation material where the photon hit map should be bound to
distance_impostor	bool	true	tells if a distance cube map impostor should be used in photon hit calculation (recommended) (the cube map will

			be bound to the first texture unit of the photon hit map material)
update_all_face	bool	false	defines if all cube map faces of the caustic cube map should be updated in a frame or only one face per frame
attenuation	float	1.0	attenuation distance of the caustic
use_triangles	bool	false	sets if triangles should be rendered into the caustic cube map instead of sprites - the two different methods need different caustic map materials: for sprites use "GTP/Caustic/CauCube_PointSprite", for triangles use "GTP/Caustic/CauCube_Triangles"
Blur_caustic_cubemap	bool	false	sets if the caustic cube map should be blurred (recommended if rendering caustic triangles)

3.1.4.4. Caustic receiver technique

This technique defines that the object will receive caustic lighting from caustic caster objects. The caustic light spots will be calculated by caustic caster's techniques. These techniques will only be updated if caustic receivers are visible, so it is the receiver technique's responsibility to trigger them. Each caustic caster's light contribution will be added in a separate pass (these passes will be inserted just after the pass in which the CausticReceiver technique is defined).

Name
CausticReceiver

Parameters

Name	Type	Default Value	Description
Max_caster_count	unsigned int	1	the maximum number of caustic casters from which this receiver can receive caustic light
vertex_program_name	string	"GTP/Basic/Shaded_VS"	the name of the vertex program that should be used in the caustic gathering passes
fragment_program_name	string	"GTP/Caustic/GatherCaustic_Cube_PS"	the name of the fragment program that should be used in the caustic gathering passes

3.1.4.5. Downsampled color cube map render technique

This technique specifies that the rendering will need a downsampled color cube map. This reduced sized cube map is created by averaging the original cube map. This downsampled cube map can easily be convolved in the final shading to achieve special effects like diffuse reflections. This technique will

automatically create a color cube map (if it is not already defined with a ColorCubeMap technique) with the given parameters.

Name
ReducedColorCubeMap

Parameters

Name	Type	Default Value	Description
update_interval	unsigned int	1	update frequency (if 0 only updates once)
start_frame	unsigned int	1	adds an offset to the current frame number to help evenly distribute updates between frames (the frame number of the first update)
resolution	unsigned int	256	cube map resolution
reduced_resolution	unsigned int	8	resolution of the downsampled cube map
texture_unit_id	unsigned int	0	the id of the texture unit state the resulting cube map should be bound to
distance_calc	bool, float	true 2.0	flag to skip cube face update if object is far away - bool value to enable calculation - float value defines the distance tolerance used in face skip
face_angle_calc	bool, float	true 2.0	flag to skip cube face update if face is negligible - bool value to enable calculation - float value defines angle tolerance used in face skip
update_all_face	bool	false	defines if all cube map faces should be updated in a frame or only one face per frame
render_self	bool	false	sets if the object should be rendered to the cube map
render_env	bool	true	sets if the environment should be rendered to the cube map
self_material	string	""	the material that should be set for the object while rendering the cube map (if no string is given the

			original materials are used)
env_material	string	""	the material that should be set for the environment while rendering the cube map (if no string is given the original materials are used)
attach_to_texture_unit	bool	true	sets if this cube map should be attach to a texture unit of the pass
layer	unsigned int	0	the layer of this cube map (multiple layers are used by shaders dealing with multiple reflections and refractions)
get_min_max	bool	0	sets if the minimum and maximum values of the cube map should be computed (it is quite a slow process as it requires texture read back to system memory - it is used to speed up linear search in cube maps while finding intersection points in multiple reflections)
Min_var_name	string	""	sets the name of the GPU fragment shader program parameter to which the minimum value should be bound to (if no string is given the variable is not bound)
Max_var_name	string	""	sets the name of the GPU fragment shader program parameter to which the maximum value should be bound to (if no string is given the variable is not bound)

3.1.4.6. Path map technique

This technique defines that the rendering of the object will add indirect lighting with the use the precomputed light path maps (see Section 6 on the Path Map preprocessor). It will create a new pass after the pass that defines the technique. The new pass will add indirect lighting to the object.

Name
PathMap

Parameters
no parameters needed

3.1.4.7. Spherical billboard technique

This is a technique for rendering particle systems with the *spherical billboard method*. This technique defines that the final rendering will need a depth map (camera space z-values) of the scene. With the use of this map scene geometry can be taken into account during final rendering of the particles, and we can eliminate billboard clipping and popping artifacts.

Name
SphericalBillboard

Parameters

Name	Type	Default Value	Description
texture_unit_id	unsigned int	1	the texture unit id where the scene depth texture will be bound

3.1.4.8. Fire Technique

This technique is similar to the spherical billboards render technique. It also defines that the rendering will need a scene depth map from the camera. The difference is that the particles will be rendered to a separate texture instead of the default frame buffer. This image can be blended with the frame buffer in a post processing step with a *compositor*. Particles are rendered into two textures simultaneously with multiple render targets, including the color buffer of the particles and the so-called heat texture. The heat texture is used to simulate heat shimmering. It stores offset values that are used at post processing. Higher offset values result in higher distortion to the image. So the shader that renders fire particles should be prepared to render into two render targets, output the color in the first and output 2D offset values in the second texture. One compositor script should use these textures and refer to them by names ILLUM_FIRE_COLOR_TEXTURE and ILLUM_FIRE_HEAT_TEXTURE. The compositor should blend the color texture with the frame buffer using the offsets stored in the heat texture.

There is also an option to set the resolution of these two textures smaller than the frame buffer. This speeds up rendering but decreases quality. This can be set with the function

OgreIlluminationManager::setFireRenderTargetSize(int size).

If *size* is two, then the resolution of these textures will be half of the frame buffer resolution. If it is three, it will be third of the frame buffer resolution, etc.

Name
Fire

Parameters

Name	Type	Default Value	Description
texture_unit_id	unsigned int	1	the texture unit id where the scene depth texture will be bound

3.1.4.9. Hierarchical Particle System Technique

This is a technique for rendering a *hierarchical particle system*. A hierarchical particle system is a particle system made of instances of a smaller particle system. It renders an image of the smaller particle system (impostor image) and multiplies this image to achieve a bigger particle system. This way less computation is needed to simulate large number of particles, while the trick is usually unnoticeable.

Name
HPS

Parameters

Name	Type	Default Value	Description
update_interval	unsigned int	1	update frequency (if 0 only updates once)
Start_frame	unsigned int	1	adds an offset to the current frame number to help evenly distribute updates between frames (the frame number of the first update)
Resolution	unsigned int	256	resolution of the impostor texture
texture_unit_id	unsigned int	0	the id of the texture unit state where the impostor texture should be bound to
Perspective	bool	true	sets if the impostor should be rendered with a perspective projection or orthogonal
particle_script	string	""	the texture unit id where the scene depth texture will be bound
Material	string	""	use this specific material for the small particle system while rendering the impostor - if empty string it will use the material defined in the particle script of the small particle system (particle_script)
vparam_radius	string	""	name of the GPU vertex program parameter where the small particle system bounding radius should be bound to - if empty string no parameter is bound
fparam_radius	string	""	name of the GPU fragment program parameter where the small particle system bounding radius should be bound to - if empty string no parameter

			is bound

3.1.4.10. Light Illumination Volume Technique

This is a technique for rendering a light illumination volume of a particle system. Light illumination volumes are used when the self shadowing of particle systems should be simulated. Each layer of the volume represents the amount of transmitted light. The current implementation uses four grayscale layers, and places these layers to the four channels of a 2D light volume texture. The GPU shader of the pass defining this technique can use this texture to shadow the particles.

Name
IllumVolume

Parameters

Name	Type	Default Value	Description
update_interval	unsigned int	1	update frequency (if 0 only updates once)
start_frame	unsigned int	1	adds an offset to the current frame number to help evenly distribute updates between frames (the frame number of the first update)
resolution	unsigned int	256	resolution of the light volume texture
texture_unit_id	unsigned int	2	the id of the texture unit state where the resulting light illumination volume texture should be bound to (in the pass defining this technique)
material	string	"Smoke_ IllumVolume"	the name of the material that is used while rendering the light volume
use_hier_system	bool	false	set this flag to true if the particle system is a hierarchical particle system
impostor_texture_unit_id	unsigned int	0	the id of the texture unit state where the impostor image of the smaller system should be bound to (in the material used while rendering the light volume) - only used if this is a hierarchical particle system
lightmatrix_param_name	string	"lightView Proj"	the name of the gpu vertex program parameter where the light matrix should be bound to (in the pass defining this technique)

3.1.4.11. Indirect texturing technique

Indirect texturing technique looks up textures to compute the texture coordinates for other textures. We use indirect texturing to obtain high resolution leaf textures for billboard cloud trees, but their application field is wider. The billboard cloud is the input geometry. The leaf impostor distribution shows where the leaves are and how they are rotated. Based on their stored values, the rotated leaf texture is looked up for color values.

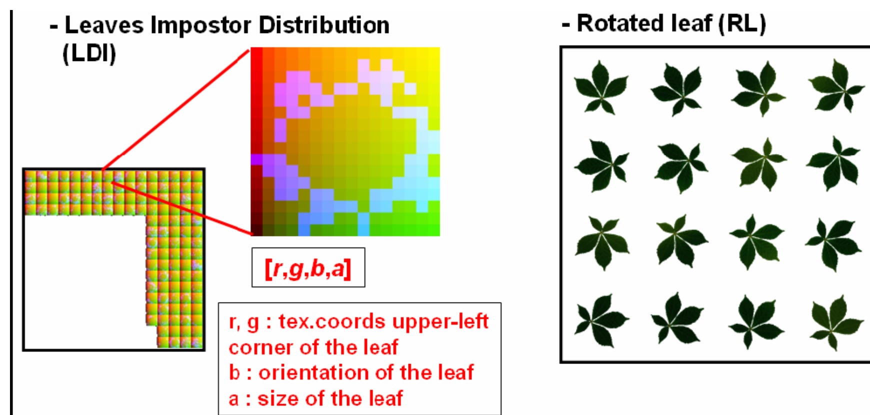

Figure 1. Leaves distribution texture atlas and rotated leaf texture atlas. These texture atlases are used in the Indirect texturing technique.

The shading context properties in Ogre3D are defined by an Ogre3D material script. The indirect texturing technique uses the following material script to setup the rendering context.

/gametools/gtp/trunk/App/Demos/Illum/IBRBillboardCloudTrees/OGRE/media/chestnut/leaves/chestnutLeavesMaterial.material

The indirect texturing material relies on a vertex shader and a fragment shader stored in the following repository folder:

/gametools/gtp/trunk/App/Demos/Illum/IBRBillboardCloudTrees/OGRE/media/general/indirectTexturingDefault_FP20.cg

/gametools/gtp/trunk/App/Demos/Illum/IBRBillboardCloudTrees/OGRE/media/general/indirectTexturingDefault_VP20.cg

3.1.4.12. Postprocessing effects

The Ogre3D engine has a built-in post-processing framework which can be used to apply effects on the final image like bloom and tone mapping. In order to support them, final rendering is not performed directly onto the screen, but to render target textures of identical dimensions. Post processing effects operate on these textures to generate the final image.

The compositor framework is driven by various compositor scripts. The textures used as render targets of the final rendering can be defined in these scripts.

```
texture scene target_width target_height PF_FLOAT16_RGBA
texture cut 256 256 PF_FLOAT16_RGBA
```

Listing 1. Compositor script fragment to set render target texture.

For every render target there is a section governing the post-processing rendering process. This is defined the same way as the rendering qualities of regular scene entities: through material scripts. Post processing can be imagined as rendering a full-screen quadrilateral with the original final rendering result as an input texture. The compositor scripts run in the order of being added to the compositor manager object.

```
CompositorManager::getSingleton().addCompositor(mWindow->getViewport(0),
                                                            "GameTools/Glow");
CompositorManager::getSingleton().setCompositorEnabled(mWindow->getViewport(0),
                                                            "GameTools/Glow", true);
CompositorManager::getSingleton().addCompositor(mWindow->getViewport(0),
                                                            "GameTools/ToneMap");
CompositorManager::getSingleton().setCompositorEnabled(mWindow->getViewport(0),
                                                            "GameTools/ToneMap", true);
```

Listing 2. Adding compositor script to the compositor manager in the application.

For the bloom effect we first have to create an image containing only the pixels with high-luminance value. Then, we have to filter this image in several passes to apply a low-pass treatment. Finally, we simply add this filtered bloom image to the existing rendering of the original scene.

```
compositor GameTools/Glow
{
  technique
  {
  texture scene target_width \\
          target_height PF_FLOAT16_RGBA
  texture cut 256 256 PF_FLOAT16_RGBA
  texture rt0 256 256 PF_FLOAT16_RGBA
  texture rt1 256 256 PF_FLOAT16_RGBA
  texture prev 256 256 PF_FLOAT16_RGBA

  target prev
  {
    input none
    only_initial on
    pass render_quad
    {
      material GameTools/PostProcBlack
    }
  }

  target scene
  {
    input previous
  }

  target cut
  {
    input none
    pass render_quad
    {
      material GameTools/GlowCut
      input 0 scene
      input 1 prev
    }
  }

  target rt0
  {
    input none
    pass render_quad
    {
      material GameTools/GlowBlurV
      input 0 cut
    }
  }

  target rt1
  {
    input none
    pass render_quad
    {
      material GameTools/GlowBlurH
      input 0 rt0
    }
  }

  target prev
  {
    input none
    pass render_quad
    {
      material GameTools/TextureCopy
      input 0 rt1
    }
  }

  target_output
  {
    input none
    pass render_quad
    {
      material GameTools/GlowAdd
      input 0 scene
      input 1 rt1
    }
  }
}
}
```

Listing 3. Compositor script to render bloom.

For the tone mapping post-processing effect we have to render a luminance map. This luminance map has to be scaled according to the appropriately filtered the local and global average luminance. Using these luminance averages the final image can be rendered in the last pass.

```
compositor GameTools/ToneMap
{
  technique
  {
  texture scene target_width \\
          target_height PF_FLOAT16_RGBA
  texture luminance 256 256 PF_FLOAT16_RGBA
  texture rt0 256 256 PF_FLOAT16_RGBA
  texture rt1 256 256 PF_FLOAT16_RGBA

  target scene
  {
    input previous
  }

  target luminance
  {
    input none
    pass render_quad
    {
      material GameTools/Luminance
      input 0 scene
    }
  }

  target rt0
  {
    input none
    pass render_quad
    {
      material GameTools/GlowBlurV
      input 0 luminance
    }
  }
  target rt1
  {
    input none
    pass render_quad
    {
      material GameTools/GlowBlurH
      input 0 rt0
    }
  }

  target_output
  {
    input none
    pass render_quad
    {
      material GameTools/ToneMap
      input 0 scene
      input 1 rt1
      input 2 luminance
    }
  }
}
}
```

Listing 4. Compositor script for tone mapping.

Figure 2. Bloom and tone mapping results.

3.1.5. Building modules with techniques in Ogre3D

3.1.5.1. Localized reflection and refraction module

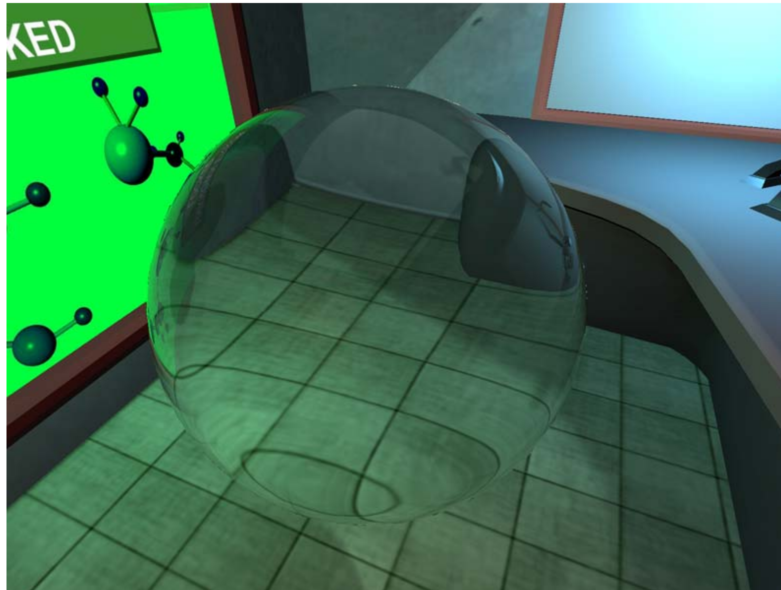

Figure 3. A reflective and refractive sphere

The localized reflection and refraction module can display reflections and refractions more accurately than classic environment mapping. The main concept of the tool is to create not only one cube map of the environment but also another one storing the distance values from the center of the cube map. The second cube map is called distance impostorcube map as it is a sampled representation of the surrounding environment. This sampled data makes it possible to search for ray-object intersections within the fragment shader without sending complex triangle data to this shader. Our fragment shader code uses a secant search algorithm to find ray-object intersection points for both reflected and refracted rays. Once a hit point is found an exact direction is given from where color values should be read from the color cube map.

Listing 2 shows the Ogre3D material script of the sphere object shown in figure 3. As it needs two cube maps (a color and a distance), two techniques should be defined: *ColorCubeMap* and *DistanceCubeMap*. This will tell the illumination manager to update these cube maps with the given frequency. In this case zero frequency means that the cube map should be refreshed only once in the first frame. Note that this does not prevent the object from changing its orientation or position. The sampling still remains valid and can give good results within a moderate range and with arbitrary orientation, only the reference point (the center of the cube map when it was last refreshed) should be known. That is why the *DistanceCubeMap* technique binds a variable named *lastCenter* - which contains this position - to the actual fragment shader.

Both *ColorCubeMap* and *DistanceCubeMap* techniques create textures that will be bound to texture unit states of the pass that defined by them (by default *ColorCubeMap* binds to the first while *DistanceCubeMap* binds to the second texture unit state).

The materials and shader codes that are related to localized reflections and refractions can be found in the repository in the following path:

\gametools\gtp\trunk\App\Demos\Illum\Ogre\Media\materials\GTPEnvMap\

```
material GTP/EnvMap/Localized_Reflector_Refractor
{
 technique
 {
  pass
  {
   IllumTechniques
   {
    RenderTechnique ColorCubeMap
    {
     update_interval          0
    }
    RenderTechnique DistanceCubeMap
    {
     update_interval          0
    }
   }
   vertex_program_ref GTP/Basic/ShadedTex_VS
   {
    param_named_auto WorldViewProj
        worldviewproj_matrix
    param_named_auto World world_matrix
    param_named_auto WorldInv inverse_world_matrix
   }
   fragment_program_ref
     GTP/EnvMap/Localized_Refraction_PS
   {
    param_named_auto cameraPos camera_position
    param_named lastCenter float3 0 0 0
    param_named sFresnel float 0.104
    param_named sRefraction float 0.6667
   }
   //Cube map for reflections and refractions
   texture_unit
   {
   }
   //Cube map of distances
   texture_unit
   {
    filtering none
   }
  }
 }
}
```

Listing 5. Material script of a glass like surface using localized reflections and refractions

3.1.5.2. Multiple reflection and refraction module

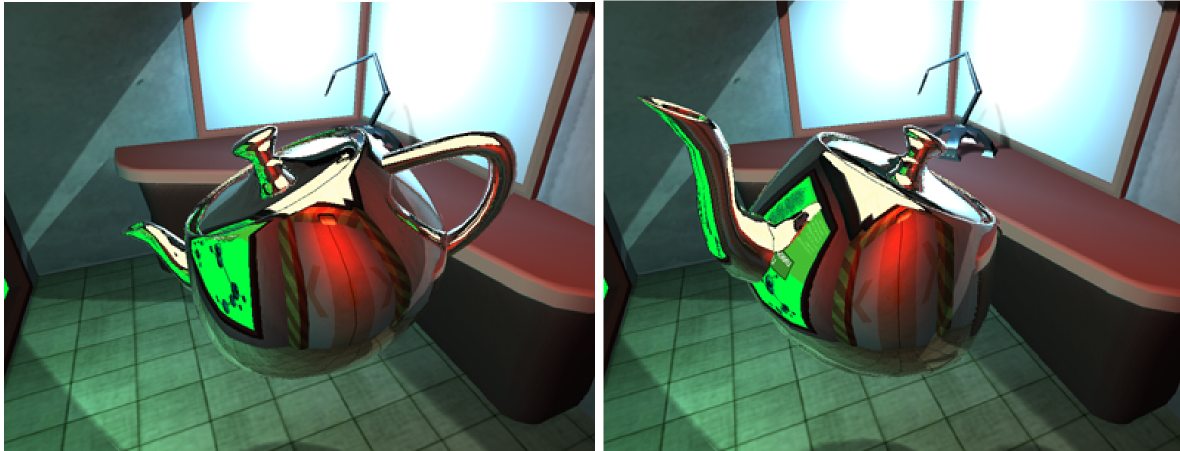

Figure 4. A reflective teapot with multiple reflections.

The multiple reflection and refraction module extends the capabilities of the reflection and refraction module. It builds on the same concepts namely the scene can be sampled by a distance impostor cube map, but it goes further as it uses multiple distance impostor cube maps. Each distance impostor samples a layer further from the reference point. An other extension is that not only the surrounding environment but the reflective object itself is sampled in the maps. This enables us to search for ray intersections even with the object itself so multiple ray bounces can be simulated.

Though different layers can be calculated with depth peeling, we used a simplified but much faster approach. We store three cube maps: one for distant objects that does not contain reflective and refractive objects – or their multiple reflections are not necessary to take into account – (this cube map is identical to the *DistanceCubeMap* used in the localized reflection and refraction tool); and two layers for the object itself. One layer stores the nearest surface points facing to the center of the object and the other stores back facing polygons. These last two layers will store not only the distance values but the normal vectors because this is a necessary information to determine outgoing ray directions form the given surface point. To the determine the reflected or refracted color after multiple ray bounces, we also need the color of the surfaces. For the distance environment we can use a color cube map just like in the localized reflection and refraction tool. For the object itself we do not need color information as the object reflects or refract light rays.

Listing 3 shows the Ogre3D material script of the teapot shown in figure 4. In this material four techniques are defined. *ColorCubeMap* that renders the environment with its own color. *DistanceCubeMap* that renders the environment with a material computing distance values. *ColorCubeMap* that renders only the object with a material that writes normal and distance values for front facing polygons, and another one which handles back facing polygons only.

The rendered cube maps are bound to the fragment shader of the material. Our fragment shaders use linear search followed by a secant search to find intersection points in the cube map layers. The materials and shaders related to multiple reflection and refraction can be found in the repository in the following path:

\gametools\gtp\trunk\App\Demos\Illum\Ogre\Media\materials\GTPAdvancedEnvMap\multibounce\

```
material GTP/MultiBounce/Reflector
{
 technique
 {
  pass
  {
   IllumTechniques
   {
    RenderTechnique ColorCubeMap
    {
     resolution        1024
     update_interval   1
     distance_calc     false
     face_angle_calc   false
     update_all_face   true
    }
   RenderTechnique DistanceCubeMap
   {
     resolution        1024
     update_interval   1
     distance_calc     false
     face_angle_calc   false
     update_all_face   true
   }
  RenderTechnique ColorCubeMap
  {
   resolution          512
   layer               1
   texture_unit_id     2
   update_interval     1
   distance_calc       false
   face_angle_calc     false
   update_all_face     true
   render_env          false
   render_self         true
   self_material GTP/MultiBounce/NormalDistanceCCW
  }
  RenderTechnique ColorCubeMap
  {
   resolution          512
   layer               2
   texture_unit_id     3
   update_interval     1
  }
     distance_calc     false
     face_angle_calc   false
     update_all_face   true
     render_env        false
     render_self       true
     self_material GTP/MultiBounce/NormalDistanceCW
    }
   }
   vertex_program_ref GTP/Basic/Shaded_CPos_VS
   {
    param_named_auto WorldViewProj
       worldviewproj_matrix
    param_named_auto World   world_matrix
    param_named_auto WorldInv inverse_world_matrix
   }
   fragment_program_ref
       GTP/MultiBounce/Reflection_PS
   {
    param_named_auto cameraPos camera_position
   }
   //Cube map of environment
   texture_unit
   {
   }
   //Cube map of environment distances
   texture_unit
   {
    filtering none
   }
   //Cube map of reflective object's normals and distances
   //back facing polygons only
   texture_unit
   {
    filtering none
   }
   //Cube map of reflective object's normals and distances
   //front facing polygons only
   texture_unit
   {
    filtering none
   }
  }
 }
}
```

Listing 6. Material script of a surface that uses multiple reflections

3.1.5.3. Caustics module

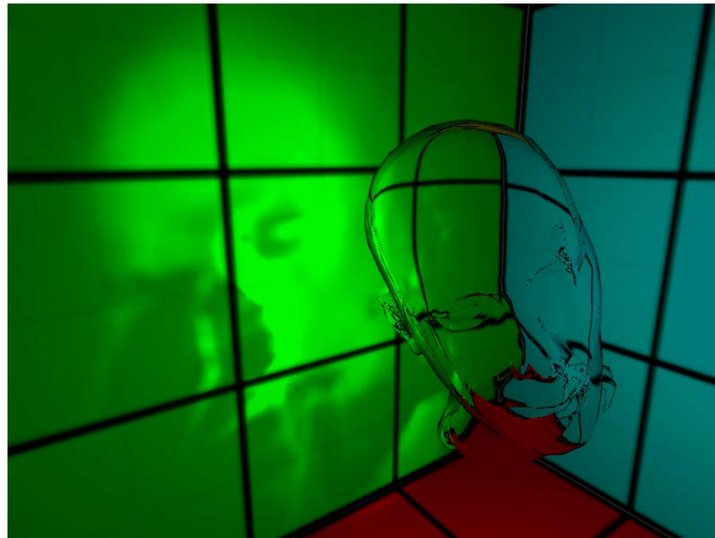

Figure 5. Refracting glass head casting caustics on a colored wall.

The main goal of the caustic module is to simulate caustic lighting effects caused by refractive objects such as glass. The caustic generation has the following steps. First the caustic caster objects are identified. For each object, a photon map is generated which stores the position of the photons that are coming from a light source refracted by the caustic caster object and arriving at a caustic receiver object. Now the photons can be drawn to the screen as point primitives or sprites but this would cause artifacts so we took an other approach. The photons are rendered to a so called caustic cube map. This cube map is similar to a cube map that is used in shadow mapping for a point light source but instead of shadowing information it stores lighting information. If the caustic cube maps are rendered, caustic receivers can use them to add additional lighting to their surface.

Listing 4 shows the Ogre3D material script of a glass head shown in figure 5. It defines the *ColorCubeMap* and *DistanceCubeMap* techniques since rendering the head will require localized refractions – just like in the localized reflection and refraction tool. The material also defines the *CausicCaster* technique that tells the illumination manager to create a photon map and a caustic cube map and refresh them with the given frequency. The technique defines the materials used in photon map and caustic cube map rendering. It also defines that a distance impostor is used in the photon map generation which makes hit point calculations more accurate (the distance impostor is bound to the first texture unit of the material used in the photon map rendering).

Listing 5 shows the Ogre material script of the caustic receiver room wall. It defines the *CausticReceiver* technique which will add additional passes – right after the pass that defined it – that will add extra lighting to the object. The number of passes depends on the number of caustic casters this receiver can receive caustics from (this can be set as a parameter of the caustics receiver technique). For each pass, the caustic cube map of the actual caster is bound to the first texture unit of the fragment shader or can be set as a parameter. The extra passes will use blending that adds the lighting stored in the caustic cube map. The shaders and materials related to caustic generation can be found in the following path of the repository:

/gametools/gtp/trunk /App/Demos/Illum/Ogre/Media/materials/GTPCaustic/

```
material GTP/Caustic/Glass
{
 technique
 {
  pass
  {
   IllumTechniques
   {
    RenderTechnique ColorCubeMap
    {
     update_interval              0
     update_all_face true
    }
   RenderTechnique DistanceCubeMap
    {
     update_interval  0
     update_all_face true
    }
   RenderTechnique CausticCaster
    {
     update_interval                    1
     photonmap_resolution               64
     distance_impostor                  true
     photon_map_material
          GTP/Caustic/PhotonMap_HitEnv
     caustic_cubemap_resolution         256
     caustic_map_material
          GTP/Caustic/CauCube_Triangles
     use_triangles                      true
     update_all_face                    true
     blur_caustic_cubemap               true
    }
   vertex_program_ref GTP/Basic/Shaded_VS
   {
    param_named_auto WorldViewProj
       worldviewproj_matrix
    param_named_auto World world_matrix
    param_named_auto WorldInv inverse_world_matrix
   }
   fragment_program_ref
     GTP/EnvMap/Localized_Refraction_PS
   {
    param_named_auto cameraPos camera_position
    param_named lastCenter float3 0 0 0
   }
   param_named sFresnel float 0.1
   param_named sRefraction float 0.8
  }
  //Cube map of colors
  texture_unit
  {
  }
  //Cube map of distances
  texture_unit
  {
    filtering none
  }
  }
 }
}
```

Listing 7. Material script of a glass object that casts caustics

```
material colorcube
{
 technique
 {
  pass
  {
   IllumTechniques
   {
    RenderTechnique CausticReceiver
    {
     max_caster_count       10
     pass_blending dest_colour one
    }
   }
   lighting off
   texture_unit
   {
     texture roomcdark.PNG
   }
  }
 }
```

Listing 8. Material script of a surface that receives caustics

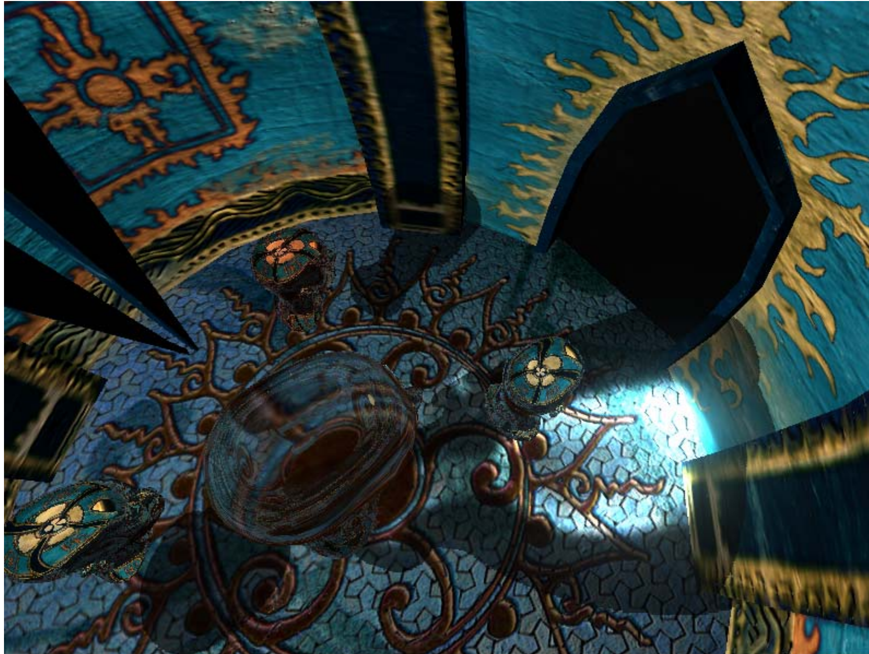

Figure 6. Caustics and environment mapping implemented in Ogre3D. The demo shows four reflecting heads with different metal shaders rotating around a refracting head which casts caustics on the walls and floor. The reflecting and refracting objects use localized environment mapping. The program uses a caustic generating method. Also a glow effect was added to the scene.

Figure 7. Reflections, refractions, and caustics in Ogre3D.

3.1.5.4. Diffuse inter-reflection module

Figure 8. Buddha illuminated by diffuse indirect lighting.

The diffuse inter-reflection module computes the lighting of an object with a cube map. It treats the pixels of the cube map as virtual light sources an cycles them adding their contribution together. This is very time consuming for a cube map with high resolution while using low resolution cube maps would speed up calculation but cause artifacts as it does not contain enough information. Our approach is to use a low resolution cube map that is a down sampled version of the high resolution cube map. This down sampled cube map will contain enough information of the environment as it stores interpolated values (each pixel can be treated as an area light source with an intensity of the average intensity if the down sampled pixels).

Listing 6 shows the Ogre material script of the diffuse Buddha. This material defines the *ReducedColorCubeMap* technique which tells the illumination manager to create a color cube map for the object (if already not created) and a down sampled version of this cube map and refresh them with the give update frequency. It also defines a *DistanceCubeMap* as the position of the virtual light sources is also needed for accurate lighting. The down sampled cube map and the distance impostor cube map are bound to the texture units of the pass that defined the techniques. The fragment shader calculates indirect illumination using disc-to-point form factor approximation cycling through each pixel of the down sampled cube map.

The shaders and materials related to diffuse indirect illumination can be found in the following path of the repository:

\gametools\gtp\trunk\App\Demos\Illum\Ogre\Media\materials\GTPAdvancedEnvMap\diffuse\

```
material GTP/Diffuse/Disc2Point
{
 technique
 {
  pass
  {
   IllumTechniques
   {
    RenderTechnique DistanceCubeMap
    {
     update_interval  1
     update_all_face  false
     distance_calc   false
     face_angle_calc  false
     resolution      128
    }
    RenderTechnique ReducedColorCubeMap
    {
     update_interval  1
     reduced_resolution 4
     resolution 128
    distance_calc false
    face_angle_calc false
    update_all_face false
   }
   }
     vertex_program_ref GTP/Basic/Shaded_VS
     {
      param_named_auto WorldViewProj
          worldviewproj_matrix
      param_named_auto WorldInv
          inverse_world_matrix
      param_named_auto World
          world_matrix
     }
     fragment_program_ref GTP/Diffuse/Disc2Point_PS
     {
      param_named_auto cameraPos camera_position
      param_named lastCenter float3 0 0 0
     }
     //Cube map of colors
     texture_unit
     {
     }
     //Cube map of distances
     texture_unit
     {
     }
    }
  }
}
```

Listing 9. Material script of diffuse inter-reflections

3.1.5.5. Spherical billboards module

Figure 9. Ogre head inside a particle system rendered with spherical billboards. Clipping and popping artifacts were removed by the spherical billboards method. The glow effect was also added.

The goal of the spherical billboards module is to eliminate billboard clipping and popping artifacts occurring while rendering planar billboards. A typical application of billboard rendering is a particle system. The spherical billboard module treats particles as spheres and computes the length a view ray travels inside them. If this length is known, the opacity of each pixel covered by a particle can be exactly calculated. To do this, the spherical billboard module needs a representation of the scene. This will be given by a depth map taken from the frame buffer's camera.

Listing 7 shows the Ogre3D material script of the particle system. The material defines the *SpericalBillboard* technique which tells the illumination manager to render a depth map from the camera, refresh it in each frame and bind it to the given texture unit state of the pass that defined the technique. The fragment program of the pass will read depth values from the depth map, calculates view ray length inside particles and the corresponding opacity. We should note here that the spherical billboard module assumes that the particle system script defines the *sprite* renderer type (which uses the *SpriteParticleRenderer* class provided by the GTP in Ogre3D changes).

The shaders for the spherical billboards module can be found in GTP_Sprite.hlsl in the following path of the repository:

\gametools\gtp\trunk\App\Demos\Illum\Ogre\Media\materials\ GTPParticles\

```
material GTP/SBB/Basic
{
 technique
 {
  pass
  {
   IllumTechniques
   {
    RenderTechnique SphericalBillboard
    {
      texture_unit_id       1
    }
   }
   scene_blend src_alpha one
   depth_write off
   depth_check off
   vertex_program_ref GTP/Particles/SB_Sprite_VS
   {
     param_named_auto worldView worldview_matrix
     param_named_auto Proj projection_matrix
     param_named_auto width viewport_width
     param_named_auto height viewport_height
   }
   fragment_program_ref  GTP/Particles/SB_Sprite_PS
   {
     param_named_auto farplane far_clip_distance
     param_named_auto nearplane  near_clip_distance
     param_named color float4 3 3 3 1
   }
   texture_unit
   {
     anim_texture smokealpha.tga 32 2.0
   }
   //scene depth texture
   texture_unit
   {
       filtering none
   }
   texture_unit
   {
       texture planck.tga
   }
  }
 }
}
```

Listing 10. Material script of spherical billboards

3.1.5.6. Fire module

Figure 10. Fire and explosions in an Ogre3D game.

The fire module is an extension of the spherical billboard tool. It also requires a depth map from the camera and uses it for the same purposes as the spherical billboard module. The main difference is that the fire module renders the particles into a double render texture instead of the frame buffer. The first texture stores the colors and the second texture stores offset values that will be used in post-processing to add heat shimmering to the scene caused by fire particles.

Listing 11 shows the Ogre3D material script of the fire particle system depicted in figure 10. The material defines the *Fire* technique that tells the illumination manager to create a depth map from the camera, refresh it in each frame and bind it to the given texture unit of the pass that defined the technique. The rendering will be redirected to a multiple render target with two targets. The fragment shader will write color information to the first and offset values to the second render target.

Listing 12 shows the compositor script and the compositor's material script used in the fire module. The compositor material refers to the fire color and offset textures by name: the color texture will always be named ILLUM_FIRE_COLOR_TEXTURE while the offset texture will be named ILLUM_FIRE_HEAT_TEXTURE.

This module was used in the demo game called "CarDriving_BME". The shaders, materials and the compositor script related to this module can be found in the repository in the following files:

\gametools\gtp\trunk\App\Games\CarDriving_BME\Media\materials\scripts\ Fire.material

\gametools\gtp\trunk\App\Games\CarDriving_BME\Media\materials\ programs\ Fire.hlsl

\gametools\gtp\trunk\App\Games\CarDriving_BME\Media\materials\scripts\ FireHeat.compositor

```
material Fire
{
 technique
 {
  pass
  {
   IllumTechniques
   {
    RenderTechnique Fire
    {
     texture_unit_id   1
    }
   }
   scene_blend src_alpha one  one_minus_src_alpha
   depth_write off
   depth_check off

   vertex_program_ref FireVS
   {
    param_named_auto worldViewProj
        worldviewproj_matrix
    param_named_auto worldView worldview_matrix
    param_named_auto Proj projection_matrix
    param_named_auto width viewport_width
    param_named_auto height viewport_height
   }
   fragment_program_ref  FirePS
    {
     param_named_auto farplane far_clip_distance
     param_named_auto nearplane near_clip_distance
    }
    texture_unit
    {
      anim_texture smokealpha.tga 32 2.0
    }
    texture_unit //scene depth texture
    {
      filtering none
    }
     texture_unit
    {
     texture planck.tga
    }
    texture_unit
    {
      texture gradient.tga
    }
    texture_unit
    {
      texture noise.tga
    }
   }
  }
}
```

Listing 11. Material script of fire

```
compositor FireHeatCompositor
{
 technique
 {
  texture scene target_width target_height
        PF_FLOAT16_RGBA
  target scene
   {
     input previous
   }
   target_output
   {
        input none

        pass render_quad
        {
          material FireHeatCompositor
          input 0 scene
        }
     }
   }
}
```

```
material FireHeatCompositor
{
 technique
 {
  pass
  {
   vertex_program_ref GameTools/PostProc1_VS
   {
   }
   fragment_program_ref FireHeatCompositorFP
   {
    param_named_auto width viewport_width
    param_named_auto height viewport_height
   }
   texture_unit //scene
   {
   }
   texture_unit
   {
    texture ILLUM_FIRE_COLOR_TEXTURE
   }
   texture_unit
   {
    texture ILLUM_FIRE_HEAT_TEXTURE
   }
  }
 }
}
```

Listing 12. Material hear shimmering

3.1.5.7. Hierarchical particle system with illumination volume module

Figure 11. Hierarchical particle system with self-shadowing.

The module called 'hierarchical particle system with illumination volume' renders a particle system composed of multiplied instances of another particle system and simulates light absorption within the media.

Listing 13 shows the material of the hierarchical particle system in figure 11. The material defines the *SphericBillboard* technique and uses a similar approach to remove billboard artifacts as the spherical billboard module (see section 3.1.5.5). It defines the *HPS* (stands for hierarchical particle system) technique that tells the illumination manager to render an impostor image of the particle system that should be multiplied (this impostor image stores color and depth information). This texture will be bound to the given texture unit of the pass that defined the technique.

The material also defines the technique *IllumVolume* which tells the illumination manager to render the particle system to be multiplied to a so called illumination volume (or light volume) texture. This texture has four "layers" stored in the four channels of the texture and each layer absorbs more and more light according to the density and current position of the particles. This texture will be bound to the desired texture unit of the pass that defined the technique.

The fragment shader of the material uses the impostor image of the particle system to color the particles; calculates exact opacity based on the scene depth map, the impostor image and position of the particle system; and finally it darkens the colors according to the absorbed light read from the illumination volume texture.

The shaders and materials related to the hierarchical particle system with illumination volume module can be found in the repository in the following path:

\gametools\gtp\trunk\App\Demos\Illum\Ogre\Media\materials\GTPParticles\

```
material GTP/HPS/Smoke_L_Depth_Illum
{
 technique
 {
  pass
  {
    IllumTechniques
    {
      RenderTechnique HPS
      {
        particle_script GTP/HPS/Smoke_Little
        perspective false
        vparam_radius baseRadius
        update_interval 1
      }
      RenderTechnique SphericalBillboard
      {
        texture_unit_id                    1
      }
      RenderTechnique IllumVolume
      {
        material GTP/HPS/Smoke_IllumVolume
        update_interval 1
        texture_unit_id 2
        resolution 128
        lightmatrix_param_name lightViewProj
      }
     }
     depth_check off
     depth_write off
     scene_blend alpha_blend
     vertex_program_ref
           GTP/HPS/Large_Depth_Illum_VS
     {
       param_named_auto worldView worldview_matrix
       param_named_auto worldViewInv
             inverse_worldview_matrix
       param_named_auto Proj projection_matrix
       param_named_auto width viewport_width
       param_named_auto height viewport_height
       param_named baseRadius float 1
      }
     fragment_program_ref
           GTP/HPS/Large_Depth_Illum_PS
     {
     }
     //impostor texture
     texture_unit
     {
     }
     //scene depth texture
     texture_unit
     {
     filtering none
     }
     //Light illumination volume texture
     texture_unit
     {
     }
    }
  }
}
```

Listing 13. Material script of hierarchical particle systems

3.1.5.8. Light path map module

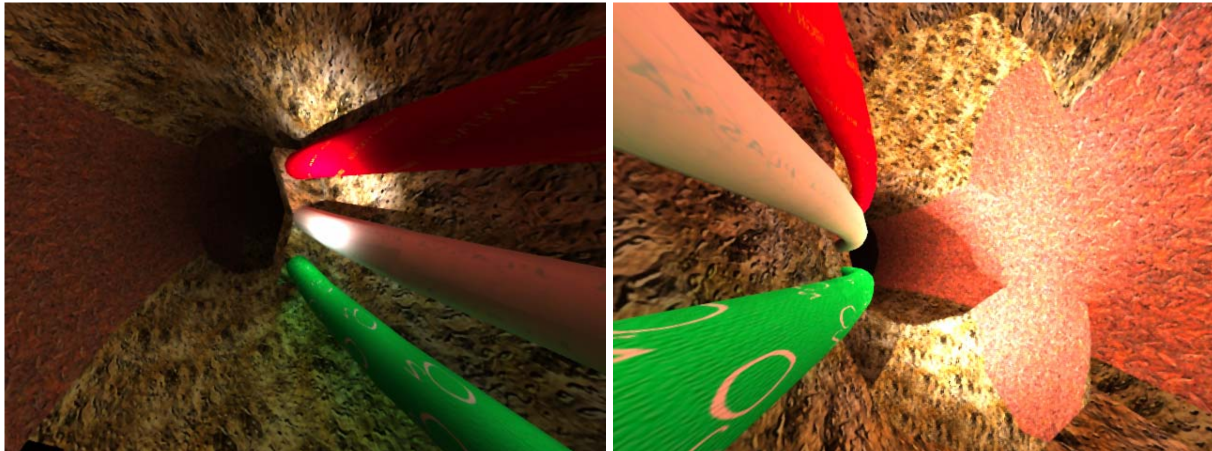

Figure 12. Real-time diffuse relighting with light path maps

The path map module adds indirect illumination based on PRM textures computed by the path map preprocessor. Listing 14 shows the Ogre3D material script of an object using the path map tool. The material defines the *PathMap* technique which creates an extra pass that adds the indirect illumination. The pre-computed PRM texture of the object is automatically bound to the new pass' first texture unit by the illumination manager. The material also uses the depth shadow mapping module (see section 463.1.5.9).

The shaders and materials related to the path map module can be found in the repository in the following path:

\gametools\gtp\trunk\App\Demos\Illum\Ogre\Media\materials\GTPPathMap

```
material PRMDemo/Base
{
 technique maintechnique
 {
  pass mainpass
  {
   IllumTechniques
   {
    RenderTechnique PathMap
    {
      pass_blending add
    }
    RenderTechnique DepthShadowReceiver
    {
     max_light_count 2
     vertex_program_name GTP/Basic/LightCPos_VS
     fragment_program_name
          GTP/Basic/SM/Dist_VSM_PS
     set_light_view true
     set_light_farplane true
     light_viewproj_param_name LightViewProj
     light_view_param_name LightView
     light_farplane_param_name lightFarPlane
     world_view_proj_param_name WorldViewProj
     world_param_name World
     pass_blending modulate
    }
   }
   vertex_program_ref GTP/Basic/ShadedTex_VS
   {
     param_named_auto WorldViewProj
          worldviewproj_matrix
     param_named_auto World world_matrix
     param_named_auto WorldInv inverse_world_matrix
   }
   fragment_program_ref
        GTP/Basic/Shaded/TexturedOneLight_PS
   {
     param_named_auto lightPos light_position 0
     param_named_auto lightDir light_direction 0
     param_named_auto lightColor light_diffuse_colour 0
     param_named_auto lightPower light_power 0
   }
   texture_unit color
   {
     colour_op replace
   }
  }
 }
}
```

Listing 14. Material script of the path map indirect illumination

3.1.5.9. Shadow mapping module

The shadow mapping module implements the shadow mapping algorithm. It implements methods that enhance the quality of traditional shadow mapping like light space perspective shadow mapping and variance shadow mapping.

Listing 16 shows the material script of a shadow receiver object. It defines the *DepthShadowReceiver* technique that will add additional passes after the pass that defined the technique. Each pass will receive shadow from a light source. The number of passes depends on the number of light sources this object can receive shadows from (can be set with a technique parameter). The shadow maps of the light sources will be bound to the first texture unit of the additional passes. The vertex and fragment shaders used in shadow receiving passes can be set as a parameter, as different light types and methods (like variance shadow mapping) require different shaders.

There are also some additional parameters that should be set using the illumination manager in the application's code. Listing 15 shows the parameters used in the scene in figure 13.

The shaders related to shadow mapping can be found in the repository in the following file:

\gametools\gtp\trunk\App\Demos\Illum\Ogre\Media\materials\GTPBasic\GTPShadowMap_PS.hlsl

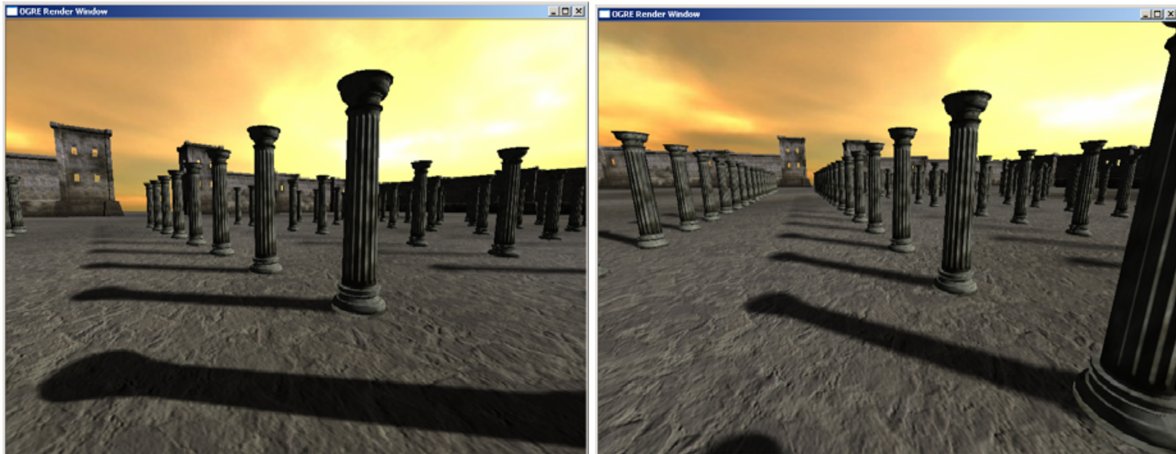

Figure 13. Soft shadows in Ogre3D

```
OgreIlluminationManager::getSingleton().setShadowMapSize(700);
OgreIlluminationManager::getSingleton().setFocusingSM(true);
OgreIlluminationManager::getSingleton().setUseLISPSM(true);
OgreIlluminationManager::getSingleton().setBlurShadowMap(true);
OgreIlluminationManager::getSingleton().setShadowMapMaterialName("GTP/Basic/DepthCCW");
```

Listing 15. Material script of the path map indirect illumination

```
material shadowReciever
{
 technique
 {
  pass
  {
   IllumTechniques
   {
    RenderTechnique DepthShadowReceiver
    {
     max_light_count 1
     vertex_program_name GTP/Basic/LightVPos_VS
     fragment_program_name
         GTP/Basic/SM/Depth_PS
     set_light_viewproj true
     set_light_view false
     set_light_farplane false
     world_view_proj_param_name WorldViewProj
     world_param_name World
     light_viewproj_param_name LightViewProj
     pass_blending modulate
    }
   }
  }
 }
```

Listing 16. Material script of the path map indirect illumination

3.1.5.10. Rain rendering module

This module introduces rain animation and rendering. It can be found in the following repository folder:

/gametools/gtp/trunk/App/Demos/Illum/Rain/

The rain animation uses a GPGPU particle system. Particles' positions are stored in textures, and updated through shaders. Rendering of the raindrops uses approximate refraction. A wide angle view of the scene is captured to a texture, which is mapped onto the drops according to the optical deviation occurring inside the drops.

This technique allows realistic integration of thousands of raindrops in a game scene. For better integration of the technique, all the main parameters influencing the appearance and total count of the drops are defined in “CommonRain.h” file, and can easily be modified.

Two rendering modes are provided: the first proposes roughly spherical raindrops, using pre-computed masks outlining the physical shape of raindrops. The second rendering mode extends the first one with retinal persistence handling. Particles are shaped into streaks, and rendered by computing a mean of a few sample overlapping positions of the moving drop creating the visual impression of a streak.

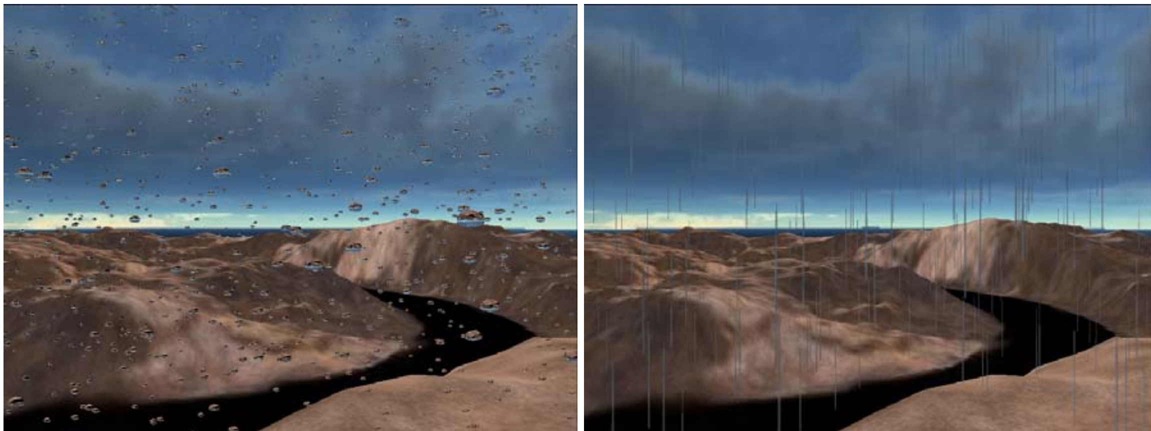

Figure 14. Rain module using regular raindrops and simulating retinal persistence.

3.1.5.11. Billboard cloud trees module

The billboard cloud tree method is based on the replacement of complex foliage by a few textured, semi-transparent quadrilaterals. We have implemented two modules based on this idea.

- *Billboard trees generated by standard texturing* uses only diffuse color texture atlas and solutions available in the Ogre3D framework. This technique is for low-end graphic cards.

  Standard texturing material:

  /gametools/gtp/trunk/App/Demos/Illum//IBRBillboardCloudTrees/OGRE/media/chestnut/leaves/chestnutLeavesMaterial.material

Standard texturing shaders:

/gametools/gtp/trunk/App/Demos/Illum//IBRBillboardCloudTrees/OGRE/media/general/

diffuseTexturing_FP20.cg

diffuseTexturing_FP20.cg

- *Billboard trees generated with indirect texturing* exploits the indirect texturing technique. It uses the rotated leaf texture atlas and the leaves distribution texture atlas in order to have high resolution rendered leaves.

The billboard tree rendering module using indirect texturing has been integrated in several demo games using Ogre3D engine.

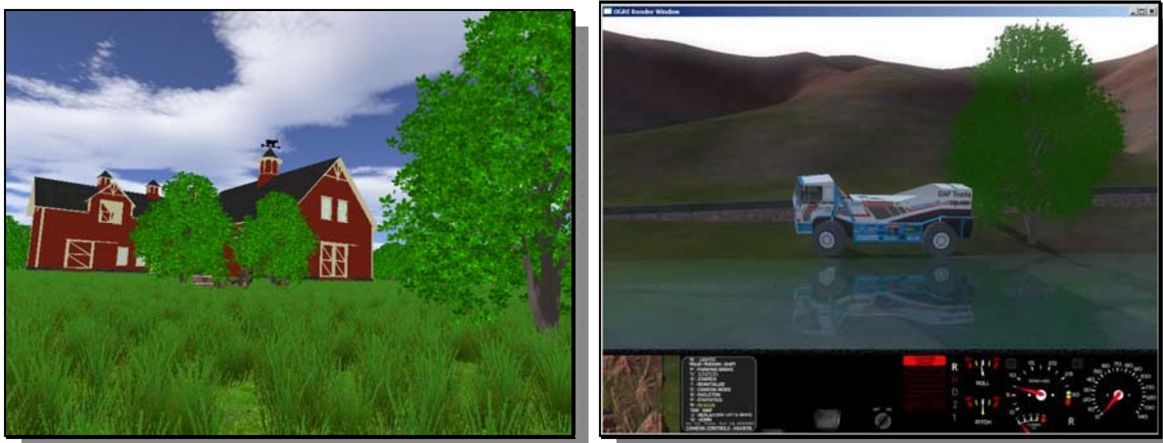

Figure 15. Right: billboard tree rendering in a demo of the public racing game Rigs of Rods with lighting and shadows. Left: a landscape scene demo with realistic lighting, using the indirect texturing technique with shadows and pre-computed obscurances for the ambient lighting.

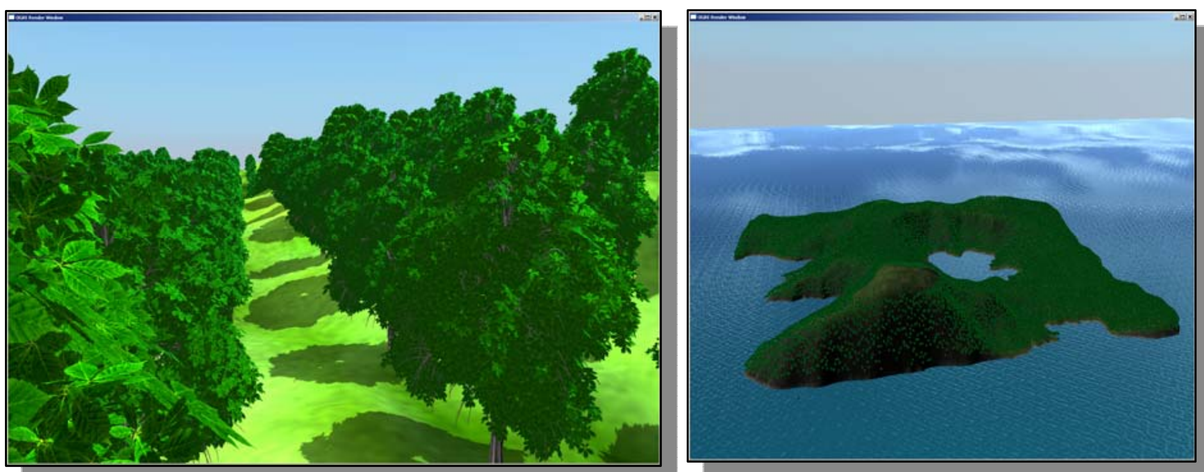

Figure 16. Left: a close view of a forest using an extended version of the indirect texturing called multi-layered indirect texturing. Right: a far view of a landscape scene demo with realistic lighting of 40000 trees.

Figure 17. Billboard trees with and without obscurances based illumination (the darker trees have been rendered with obscurances, the lighter ones without it).

3.2. SHARK3D VERSION OF ILLUMINATION MODULES

Shark3D terminology is somewhat different from what we have used in the general description or the Ogre3D version of the illumination modules. The description about rendering properties assigned to objects is not referred to as a material, but as a *shader*. For distinction, programs running on the vertex and fragment processors of the graphics card are called *GPU programs*. The implementation building block, instead of the technique, will be the *shader component*. Nevertheless, they serve the same role: assigned to objects, they encapsulate the rendering passes, which may produce resources for other components, or perform final rendering to the screen. However, unlike techniques, shader components do not include fixed GPU programs. The same shader component with different GPU programs can be the equivalent of different techniques.

GPU programs are written in HLSL, shader components in C++, and the composition of a complete shader uses the Snake language. A shader script is assigned to every actual or abstract object in the scene (meshes, light sources, post processing effects).

3.2.1. Implemented shader components of the Shark3D version

The low level building blocks are the shader components which are written in C++. In the following we detail the components that implement various rendering techniques.

3.2.1.1. eng_shader_std.paintmesh

This component is a core Shark3D component, and it has not been implemented as a part of the WP. However, it is listed here for the sake of completeness, as several GTP shader components rely on ***paintmesh*** to render objects. Typically, a GTP component generating a resource texture for later GTP components will trigger ***paintmesh*** with appropriate GPU programs to render scene objects. See Section 3.2.3 for how this is done in practice.

3.2.1.2. eng_shader_special.worldmat

This component writes out the actual world matrix into a parameter. This is to replace the original malfunctioning Shark3D component of the same purpose.

Parameters:

Name	Description
env	Shader environment object. Usually, this parameter should be "shaderenv".
methods	Methods are used internally within a shader to execute different rendering passes and other kinds of operations.
worldmat_var	The resulting matrix is stored in the variable of this name. Must be of type mat4x4f.
inverse	Sets if the matrix should be inverted.

```
{
        ident "gtp_shader.worldmat"
        param
        {
                env "shaderenv"
                methods "main_method"
                worldmat_var "worldMat"
        }
}
```

Listing 17. Shader script chunk for eng_shader_special_worldmat.

3.2.1.3. gtp_shader.causticcube

This component, with the appropriate GPU programs (see ***gtp_caustic_cubemap_point*** in Section 3.2.2), renders the caustic caster cube map from a photon hit texture. The component itself only generates a point primitive for every pixel of the photon hit texture. This is done six times, in order to render to the six faces of the caustic caster cube map. When rendering, the vertex program will move the generated primitives to the appropriate hit positions, and the fragment program will render photon sprites. If triangle rendering is used instead of point rendering, this component generates a grid.

Parameters:

Name	Description
env	Shader environment object. Usually, this parameter should be "shaderenv".
methods	Methods are used internally within a shader to execute different rendering passes and other kinds of operations.
rank	Rank of the shader in the rendering sequence. Shaders having a smaller rank are rendered first
iter_start	Iteration start value.
iter_end	Iteration end value.
iter_step	Iteration step.
iter_width_scale	Scale the image by this scale each iteration.
iter_height_scale	Scale the image by this scale each iteration.
destprop_antialias	Render the filtering with antialiasing
width_scale	Requested final horizontal scaling factor.
height_scale	Requested final vertical scaling factor.
passinfo_var	This variable is a name for the float4-vector containing information about the

	iteration. The first component of the vector contains the current iteration number, the second the total number of iterations.
shaderprog_ident	Shader program used for the smoothing iterations.
shaderprog_param_array	Defines which variables are passed to the shader program
src_array	Source array.
des_array	Destination array.
update_interval	The interval between two cube map updates.
start_frame	The first update was performed in this frame.
update_all_faces	Update the whole cube map or only the next face.
use_points	Sets if triangles (instead of sprites) should be rendered into the caustic cube map.
photon_count	The resolution of the photon map.

```
{
        ident "gtp_shader.causticcube"
        param
        {
                env "shaderenv"
                methods "caustic_cast_method"
                rank -7700
                iter_start 0
                iter_end 0
                iter_step 0
                width_scale 1.0
                height_scale 1.0
                destprop_antialias 1
                passinfo_var ""
                src_array "photonmap"
                update_interval 1
                update_all_face 1
                start_frame 0
                photon_count 32
                use_points 1
                dest_array
                {
                        from_var "cauCubeMap"
                        to_var "cauCubeMap"
                        texprop_restrusage 0
                        texprop_depth 0
                        texprop_float 0
                        texprop_comp 4
                        texprop_bitspercomp 8
                        samplermode_filter 0
                        samplermode_clamp 1
                }
        shaderprog_ident
        "levelutil/shader/prog/gtp_caustic_cubemap_point.s3d_shaderprog_run"
        shaderprog_param_array
        {
        }
}
```

Listing 18. Shader script chunk for gtp_shader_causticcube.

3.2.1.4. gtp_shader.createtex

This component can be used to generate a new texture in the shader. This is necessary to create render target textures which can be kept over several frames or during the entire lifetime of an object. This shader component should only be invoked once for every texture to be generated.

Parameters:

Name	**Description**
env	Shader environment object. Usually this parameter should be "shaderenv".
methods	Methods are used internally within a shader to execute different rendering passes and other kinds of operations.
texprop_rendertarget	Use this texture as render target.
texprop_cube	Cube map. Mutually exclusive with texprop_volume.

texprop_volume	3d texture. Mutually exclusive with texprop_cube.
texprop_mipmap	If not set, there are no mipmaps. If set, use mipmaps down to the smallest possible size.
texprop_float	Use float representation if possible.
texprop_signed	Use signed representation if possible.
texprop_restrusage	Request a texture for "restricted usage".
texprop_compressed	Allow compression of this texture.
texprop_dynamic	Prefer fast texture upload to fast rendering. For example, this may disable texture swizzling.
texprop_alpha_mask	Alpha value is only zero and one.
texprop_comp	The number of components in the texture.
texprop_bitspercomp	The number of bits in each component.
width	The width of the texture.
height	The height of the texture.
depth	The color depth of the texture.
texchan_var	Variable where texture is being stored in for use by shader components within this group.
samplermode_clamp	Clamp the texture in the sampler.
samplermode_filter	Filter the texture in the sampler.

```
{
        ident "gtp_shader.createtex"
        param
        {
                env "shaderenv"
                methods "init_method"
                texchan_var "photonmap"
                width 32
                height 32
                texprop_rendertarget 1
                texprop_float 1
                texprop_comp 4
                texprop_bitspercomp 32
                samplermode_clamp 1
                samplermode_filter 0
        }
}
```

Listing 19. Shader script chunk for gtp_shader_createtex.

3.2.1.5. gtp_shader.cubetexfilter

This component renders a full screen quad for each face of the cube map.

Parameters:

Name	Description
env	Shader environment object. Usually this parameter should be "shaderenv".
methods	Methods are used internally within a shader to execute different rendering passes and other kinds of operations.
rank	Rank of the shader in the rendering sequence. Shaders having a smaller rank are rendered first.
shaderprog_ident	Shader program used for the smoothing.
shaderprog_param_array	Defines which variables are passed to the shader program.
src_array	Source array.
dest_array	Destination array.
update_interval	The interval between two filtering step.
start_frame	The first update performed in this frame.
update_all_face	Update the whole cube map or only the next face.

```
{
	ident "gtp_shader.cubetexfilter"
	param
	{
		env "shaderenv"
		methods "main_method"
		rank -8000
		iter_start 0
		iter_end 0
		iter_step 0
		width_scale 1.0
		height_scale 1.0
		destprop_antialias 0
		passinfo_var ""
		src_array "dist"
		update_interval 0
		dest_array
		{
			from_var "filtereddist"
			to_var "filtereddist"
			texprop_restrusage 0
			texprop_depth 0
			texprop_float 0
			texprop_comp 4
			texprop_bitspercomp 0
			samplermode_filter 1
			samplermode_clamp 1
		}
	shaderprog_ident
	"levelutil/shader/prog/gtp_reduce_cubemap.s3d_shaderprog_run"
	shaderprog_param_array
	{
	}
	}
}
```

Listing 20. Shader script chunk for gtp_shader_cubetexfilter.

3.2.1.6. gtp_shader.envmap

This component renders an environment map by rendering six images of the surrounding environment onto the six faces of a cube map. Whether the faces of this cube should be aligned according to the axes of the world or view coordinate system can be selected. The user can also define the update frequency of the cube map, determining in every how-many-frames it has to be re-rendered.

What information will actually be rendered into the cube map depends on the ***paintmesh*** shader component in the shaders of rendered meshes. The environment map shader component will just trigger them to draw the meshes.

Parameters:

Name	**Description**
Env	Shader environment object. Usually this parameter should be "shaderenv".
Methods	Methods are used internally within a shader to execute different rendering passes and other kinds of operations.

rank	Rank of the shader in the rendering sequence. Shaders having a smaller rank are rendered first.
proj_neg_z	The near clipping plane.
proj_pos_z	The far clipping plane.
max_ext	Maximal texture width and height.
ext_dist_scale	Distance factor.
max_recursion	Maximal number of mirror recursions. For a single reflection, use 1.
enum_trigger	Trigger used for rendering the mirror image.
mesh_var	Name of the variable containing the mesh which is rendered.
destprop_antialias	Render the filtering with anti-aliasing.
dest_array	Destination array.
update_all_faces	Update the whole cube map or only the next face.
world_space	If set the cube map is generated in world space instead of view space.
update_interval	The interval between two filtering step.
start_frame	The first update performed in this frame.
last_center	The name of the output variable to store the last center position of the cube map.

```
{
        ident "gtp_shader.envmap"
        param
        {
            env "shaderenv"
            methods "main_method"
            rank -90000
            max_ext 256
            max_recursion 2
            ext_dist_scale 20.0
            proj_neg_z 0.1
            proj_pos_z 100.0
            enum_trigger "mirror_trigger"
            destprop_antialias 0
            mesh_var "mesh"
            update_interval 1
            update_all_face 0
            start_frame 1
            last_center_var "lastCenter"
            world_space 1
            dest_array
            {
                from_var "envmap0"
                to_var "envmap0"
                texprop_restrusage 0
                texprop_depth 0
                texprop_comp 3
                texprop_float 0
                samplermode_filter 0
                samplermode_clamp 1
            }
        }
    }
```

Listing 21. Shader script chunk for gtp_shader_envmap.

3.2.1.7. gtp_shader.focusedprojmat

This component creates a projection matrix to help focusing on a given object. When rendering a photon map for caustics, this projection matrix is helpful to set the camera looking tightly at the caustic caster object.

Parameters:

Name	Description
env	Shader environment object. Usually this parameter should be "shaderenv".
methods	Methods are used internally within a shader to execute different rendering passes and other kinds of operations.
destmat_var	Name of the variable containing the result matrix.
mesh_var	Name of the variable containing the mesh which is rendered.
centerpoint_var	The center of the focused object.

```
{
        ident "gtp_shader.focusedprojmat"
        param
        {
            env "shaderenv"
            methods "caustic_cast_method"
            mesh_var "mesh"
            destmat_var "lightViewProj"
            centerpoint_var "light_cenrange"
        }
    }
```

Listing 22. Shader script chunk for gtp_shader_focusedprojmat.

3.2.1.8. gtp_shader.mainenum

This component enumerates the objects which are visible from the camera into a variable. This is useful if an object needs to trigger potentially all other objects in the scene. For instance, a caustic caster object, after rendering its caustic caster cube map, triggers potential caustic receivers to render caustics.

Parameters

Name	Description
env	Shader environment object. Usually this parameter should be "shaderenv".
methods	Methods are used internally within a shader to execute different rendering passes and other kinds of operations.
coll_var	Name of the output variable to contain the list of the visible objects from the camera.

```
{
        ident "gtp_shader.mainenum"
        param
        {
            env "shaderenv"
            methods "main_method"
            coll_var "recievers"
        }
}
```

Listing 23. Shader script chunk for gtp_shader_mainenum.

3.2.2. GPU programs in Shark3D

The core that makes a rendering shader component operational is the GPU program. The Shark3D *shader program* files define the vertex and fragment programs and their parameters for the target platform. The shader program can contain multiple definitions for multiple environments (for example

DirectX9/HLSL and OpenGL/glsl), with links to the appropriate GPU shader. Our shader programs contain only the DirectX9 definition as target environment.

In the following we list the shader programs and the typical shader components from which they are invoked.

- gtp_reduce_cubemap
  - Filters the cube map into a cube map of reduced resolution.
  - Used in: ***gtp_shader.cubetexfilter*** (see 3.2.1.5)
  - gtp_reduce_cubemap_d3d9_hlsl_vs2x0
  - gtp_reduce_cubemap_d3d9_hlsl_ps3x0
- gtp_envmap
  - Renders a reflective, metallic object using the approximate Fresnel method. It takes a color and distance cube map as inputs, and performs localized lookup for approximate ray-tracing reflections.
  - Used in: ***eng_shader_std.paintmesh*** triggered from ***gtp_shader.envmap*** (see 3.2.1.6)
  - gtp_envmapSimple_d3d9_hlsl_vs1x1
  - gtp_envmapSimple_d3d9_hlsl_ps1x1
- gtp_distance_impostor
  - Stores distances from the camera. This is useful for rendering a distance cube map.
  - Used in: ***eng_shader_std.paintmesh*** triggered from ***gtp_shader.envmap*** (see 3.2.1.6)
  - gtp_distance_impostor_d3d9_hlsl_vs3x0
  - gtp_distance_impostor_d3d9_hlsl_ps3x0
- gtp_diffuse
  - Gathers indirect illumination with the use of the downsampled cube map.
  - Used in: ***eng_shader_std.paintmesh*** during final rendering.
  - gtp_diffuse_d3d9_hlsl_vs2x0
  - gtp_diffuse_d3d9_hlsl_ps3x0
- gtp_caustic_receive
  - Gathers caustic lighting from a caustic cube map, and renders it onto the screen, compositing caustic lights with the existing scene.
  - Used in: ***eng_shader_std.paintmesh*** triggered from ***eng_shader_std.collexec***, with a collection of objects populated by ***gtp_shader.mainenum*** (see 3.2.1.8)
  - gtp_caustic_recieve_d3d9_hlsl_vs2x0
  - gtp_caustic_recieve_d3d9_hlsl_ps2x0
- gtp_caustic_cubemap_point
  - Generates the caustic cube map. Every vertex is assumed to be a point primitive. The vertex program moves the primitives to the positions read from an input photon hit texture. The pixel program renders the caustic snippet sprites.
  - Used in: ***gtp_shader.causticcube*** (see 3.2.1.3)
  - gtp_caustic_cubemap_point_d3d9_hlsl_vs3x0
  - gtp_caustic_cubemap_point_d3d9_hlsl_ps3x0
- gtp_cau_photonmap
  - Generates the photonmap, the texture that contains photon hit positions. It stores photon hits positions in cube map space.
  - Used in: ***eng_shader_std.paintmesh*** triggered by a light source, projection matrix set as returned by ***gtp_shader.focusedprojmat*** (see 3.2.1.7)
  - gtp_cau_photonmap_d3d9_hlsl_vs3x0

  - gtp_cau_photonmap_d3d9_hlsl_ps3x0
- gtp_gen_shmap_indirect_texturing
  - Generates the shadow map for the billboard trees
  - Used in: ***eng_shader_std.paintmesh*** in final rendering.
  - gtp_gen_shmap_indirect_texturing_d3d9_hlsl_vs2x0
  - gtp_gen_shmap_indirect_texturing_d3d9_hlsl_ps2x0
- gtp_indirect_texturing_shmap
  - Renders the billboard trees with shadow mapping, using the shadow map generated by ***gtp_gen_shmap_indirect_texturing***
  - Used in: ***eng_shader_std.paintmesh*** in final rendering.
  - gtp_indirect_texturing_shmap_d3d9_hlsl_vs2x0
  - gtp_indirect_texturing_shmap_d3d9_hlsl_ps2x0

3.2.3. Building modules with shader components in Shark3D

With all the shader components and GPU programs available, the Shark3D shaders realizing illumination modules can be composed in the Shark3D Snake script language, using the chucks invoking the components. The shader scripts are directly assigned to the objects and control the rendering process.

3.2.3.1. Localized reflection module

Localized reflections are based on environment maps augmented by distance information: the distance impostors. The environment map of its surroundings has to be generated for the reflective, metallic object. Then, the object can be rendered to the screen by performing iterative lookups to find reflected where reflected eye rays hit the environment, delivering approximate ray-tracing results.

The localized reflections module is realized using two shaders. The ***ordinary*** shader is used for every object in the scene. Whenever an object has to be rendered to environment cube map with distance information, it will receive the ***gtp_distance_impostor*** trigger. The shader responds to the trigger by running a ***paintmesh*** component with the ***gtp_distance_impostor*** shader program, rendering distances.

```
{
    triggers "gtp_distance_impostor_trigger"
    method "gtp_distance_impostor_method"
}
```

Listing 24. Distance impostor trigger.

```
        {
            ident "eng_shader_std.paintmesh"
            param
            {
                env "shaderenv"
                methods "gtp_distance_impostor_method"
                rank 12000
                insp_ident "insp_shader_mesh"
                cull_mode "back"
                depth_test "less_equal"
                depth_write 1
                mesh_var "mesh"
                texchan_var_array
                tex_attr_var_array "attr0"
                attrmat_var_array ""
                use_vertex_bone_wgh 1
                use_vertex_bone_subscr 1
                use_vertex_point 1
                color 1.0 1.0 1.0
                shaderprog_ident
                    "levelutil/shader/prog/gtp_distance_impostor.s3d_shaderprog_run"
                shaderprog_param_array
            }
        }
```

Listing 25. Paintmesh component in ordinary shader for localized reflections

The second shader, ***ordinary_copper,*** is the one which is assigned to the metallic object. Its tasks are to create the color and distance cube map textures once, administer their update by triggering other objects to be rendered into them, and render the object onto the screen with localized reflections. The cube maps are created using the ***gtp_shader.createtex*** component.

```
        {
            ident "gtp_shader.createtex"
            param
            {
                env "shaderenv"
                methods "init_method"
                texchan_var "envmap0"
                width 128
                height 128
                texprop_rendertarget 1
                texprop_cube 1
                texprop_float 0
                texprop_comp 4
                texprop_bitspercomp 8
                samplermode_clamp 0
                samplermode_filter 1
            }
        }
```

Listing 26. Creating the cube map.

To render the environment to this cube map, you have to use the ***gtp_shader_envmap*** component. It has to send the ***gtp_distance_impostor_trigger*** to all the objects in the scene.

```
        {
            ident "gtp_shader.envmap"
            param
            {
                env "shaderenv"
                methods "main_method"
                rank -90000
                max_ext 128
                max_recursion 1
                ext_dist_scale 20.0
                proj_neg_z 0.1
                proj_pos_z 100.0
                enum_trigger "gtp_distance_impostor_trigger"
                destprop_antialias 0
                mesh_var "mesh"
                update_interval 1
                update_all_face 0
                start_frame 1
                last_center_var "lastCenter"
                world_space 1
                dest_array
                {
                    from_var "dist"
                    to_var "dist"
                    texprop_restrusage 0
                    texprop_depth 0
                    texprop_comp 1
                    texprop_float 1
                    samplermode_filter 0
                    samplermode_clamp 1
                }
            }
        }
```

Listing 27.Rendering the distance cube map.

As our final rendering component works with world space coordinates to find localized reflections, we need to compute the world matrix using the ***gtp_shader.worldmat*** component.

```
        {
            ident "gtp_shader.worldmat"
            param
            {
                env "shaderenv"
                methods "main_method"
                worldmat_var "worldMat"
            }
         }
```

Listing 28. Acquiring the world space transformation.

The final rendering component is the ***eng_shader_std.paintmesh*** component with the localized environment mapping shader program ***gtp_shader.envmap***.

```
        {
            ident "eng_shader_std.paintmesh"
            param
            {
                env "shaderenv"
                methods "main_method"
                rank 12000
                insp_ident "insp_shader_mesh"
                cull_mode "back"
                depth_test "less_equal"
                depth_write 0
                mesh_var "mesh"
                texchan_var_array "envmap0" "dist"
                tex_attr_var_array "attr0"
                attrmat_var_array ""
                use_vertex_bone_wgh 1
                use_vertex_bone_subscr 1
                use_vertex_point 1
                use_vertex_normal 1
                use_vertex_coloralpha 0
                shaderprog_ident
                    "levelutil/shader/prog/gtp_envmap.s3d_shaderprog_run"
                shaderprog_param_array
                {
                    src_var "lastCenter"
                    dest_progvar "lastCenter"
                }
                {
                    src_var "worldMat"
                    dest_progvar "worldMat"
                }
                {
                    src_var "worldMatIT"
                    dest_progvar "worldMatIT"
                }
            }
        }
```

Listing 29. Final rendering component

Figure 18. Room with a diffuse horse reflected on a metal sphere.

3.2.3.2. Diffuse reflection module

To create a diffuse shaded object you have to render a color and a distance cube map from the object the same way we did for metallic reflections. During final rendering, these environment maps have to be convolved with the diffuse reflection distribution function to get their contribution to the illumination of the diffuse surface. To make this real-time, this is performed on reduced size, pre-filtered cube maps. This is implemented in the ***ordinary_diffuse*** shader, using the ***gtp_shader.cubetexfilter*** component.

```
{
            ident "gtp_shader.cubetexfilter"
            param
            {
                env "shaderenv"
                methods "main_method"
                rank -8000
                iter_start 0
                iter_end 0
                iter_step 0
                width_scale 1.0
                height_scale 1.0
                destprop_antialias 0
                passinfo_var ""
                src_array "dist"
                update_interval 0
                dest_array
                {
                    from_var "filtereddist"
                    to_var "filtereddist"
                    texprop_restrusage 0
                    texprop_depth 0
                    texprop_float 0
                    texprop_comp 4
                    texprop_bitspercomp 0
                    samplermode_filter 1
                    samplermode_clamp 1
                }
                shaderprog_ident
                    "levelutil/shader/prog/gtp_reduce_cubemap.s3d_shaderprog_run"
                shaderprog_param_array
                {
                }
            }
        }
```

Listing 30. Reducing the cube map in the diffuse shader.

Final rendering of the reflective diffuse object is done with a simple ***paintmesh*** component again, but setting the ***gtp_diffuse*** shader program.

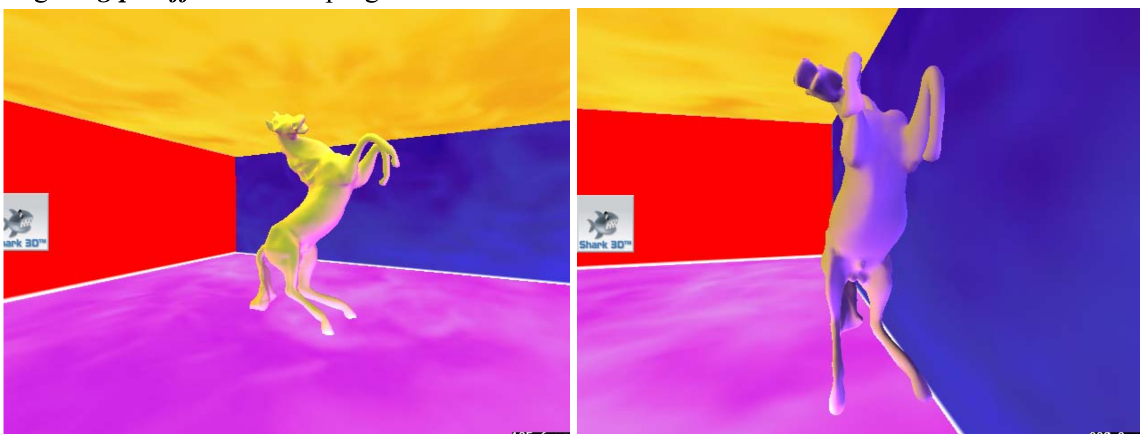

Figure 19. Diffuse horse in a test scene.

3.2.3.3. Caustics module

The communication chain of the caustics module is the most complicated. It starts with the light source. The shader of the light source should find nearby caustic casters, and trigger them to perform a chain of operations:

- A caustic caster should set up a render camera at the light source, focused on the caustic caster object itself.
- Using this setup, it should render the photon hit positions texture, a texture resource containing the cube map coordinates of refracted, exiting photons.
- Then, the caustic caster cube map has to be rendered, by splatting snippets to photon hit locations onto all six cube map faces.
- The caustic caster has to gather all visible objects.
- It has to send them the trigger to receive caustics.
- Caustic receivers have to render themselves with light from the caustic caster map projected onto them.

Triggering caustic casters from the light is realized by adding an ***eng_shader.collexec*** chunk to the light shader, sending **caustic_cast_trigger** to objects within range. This is done in our ***caustic_caster_light*** shader.

```
        {
             ident "eng_shader_std.collexec"
             param
             {
                 env "shaderenv"
                 methods "main_method"
                 coll_var "casters"
                 rank -8000
                 exec_trigger "caustic_cast_trigger"
                 exec_param_array
                 {
                     src_var "light_cenrange"
                     dest_extvar "ext_light_cenrange"
                 }
                 {
                     src_var "recievers"
                     dest_extvar "recievers"
                 }
             }
         }
```

Listing 31. Triggering the visible caustic caster objects from the light.

As defined in the shader ***caustic_caster***, the triggered caustic caster creates a photon map. It uses ***gtp_shader.focusedproj*** to compute projection matrix, then renders everything by triggering a ***paintmesh*** chunk with the ***gtp_cau_photonmap*** shader program. From the photon map, it generates the caustic cube map using the component ***gtp_shader.causticcube***.

```
{
            ident "gtp_shader.causticcube"
            param
            {
                env "shaderenv"
                methods "caustic_cast_method"
                rank -7700
                iter_start 0
                iter_end 0
                iter_step 0
                width_scale 1.0
                height_scale 1.0
                destprop_antialias 1
                passinfo_var ""
                src_array "photonmap"
                update_interval 1
                update_all_face 1
                start_frame 0
                photon_count 32
                use_points 1
                dest_array
                {
                    from_var "cauCubeMap"
                    to_var "cauCubeMap"
                    texprop_restrusage 0
                    texprop_depth 0
                    texprop_float 0
                    texprop_comp 4
                    texprop_bitspercomp 8
                    samplermode_filter 0
                    samplermode_clamp 1
                }
                shaderprog_ident
                    "levelutil/shader/prog/gtp_caustic_cubemap_point.s3d_shade"
                    & "rprog_run"
                shaderprog_param_array
            }
        }
```

Listing 32. Generating caustic cube map.

With the caustic cube map ready, the caster sends the ***caustic_receive_trigger*** triggers to the receivers.

```
{
    ident "eng_shader_std.collexec"
    param
    {
        env "shaderenv"
        methods "main_method"
        coll_var "recievers"
        rank 13500
        exec_trigger "caustic_recieve_trigger"
        exec_param_array
        {
            src_var "lastCenter"
            dest_extvar "lastCenter"
        }
        {
            src_var "cauCubeMap"
            dest_extvar "cauCubeMap"
        }
    }
}
```

Listing 33. Triggering the receivers.

As the final step, the caustic receiver objects, being triggered by the casters, execute a ***paintmesh*** chunk with the ***gtp_caustic_recieve*** shader program, to render themselves with the caustic cube map lighting projected onto them.

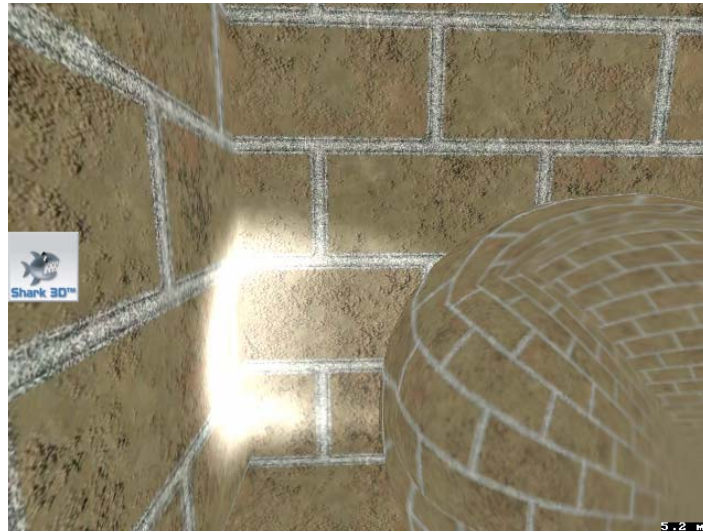

Figure 20. Caustics, reflection and shadows.

3.2.3.4. Billboard cloud trees module

The billboard tree technique has been integrated in a Shark3D using depth shadow mapping and realistic lighting.

Figure 21. The billboard tree technique in a Shark3D demo with hundreds of trees.

4. FILE FORMATS

In order to facilitate data exchange between preprocessors and engine-integrated modules, as well as to be able to model and import complete scenes, a file interface had to be specified. The core of this is the *level description*. It consists of a text file (called the *level file*), referencing *mesh files*, which are typically in the Ogre3D XML format. The level file describes where instances of meshes (called *entities*) are located, and various parameters governing preprocessing of meshes and entities. Preprocessors operate on these inputs. As Ogre3D mesh files do not include shading information other than the names of materials, and parsing all the material scripts in a standalone preprocessor is not feasible, basic material properties are also stored in a file called the *material file*. This lists basic textures or colors that can be used in preprocessors.

Preprocessors output data in similar format, plus special technique-related information. They might process meshes into other meshes, augment level files with mesh or entity-specific information, and produce texture files.

The following files can be generated by Maya or by the preprocessors.

4.1. COLLADA FILE [MEDIA*.DAE]

Collada scene file format is used by the *Billboard Tree Preprocessor* to export the simplified billboard cloud meshes to the standard DCC tools in order to do the scene composition. The Collada file format is good as an intermediate format for transporting general data from one digital content creation (DCC) tool to another. In the billboard cloud tree technique it is only used to export the geometric information.

4.2. LEVEL FILE [MEDIA*.LEVEL]

Lists meshes and entities. Every entry starts with the keyword ‘mesh’ or ‘entity’, followed by a unique name, and, between curly braces, a set of attributes.

Mesh attributes:

- ogrefile — Binary Ogre3D mesh file name.
- ogreXMLfile — XML Ogre3D mesh file name.
- xfile — DirectX mesh file name. (alternative to Ogre3D meshes)
- pathmapresolution — PRM tile resolution for Path Map generation.
- divide — Number of mesh segments to be created by the preprocessor.

Entity attributes

- mesh — mesh name
- transformation — modeling transformation as a 4x4 matrix
- pathmapclusters — number of clusters relevant to subentities of this entity
- pathmapfile — output texture file name stub

4.3. MATERIAL FILE [MEDIA*.MATERIALS]

XML file that describes Ogre3D materials for the Path Map preprocessor. For every material name, a texture name or a solid color is given. Between the root <materials> and </materials> tags, material entries are listed. The 'name' attribute specifies the name of the material, which is used in the Ogre3D mesh files. The 'texture' attribute gives the image file used as a primary texture, and the 'color' attribute specifies a solid color.

4.4. MESH FILES [MEDIA*.MESH.XML, PROCESSEDMESHES*.MESH.XML, *.MESH]

Ogre3D meshes are used in the Path Map preprocessor, the Billboard cloud preprocessor, the Obscurances preprocessor, and they are loaded into the Ogre3D engine when running the integrated modules. Meshes are both input and output. The Ogre3D XML converter tool bundled with Ogre3D can be used to convert between XML and binary formats. A batch file invoking the Ogre3D XML converter is also provided along the Maya exporter scripts (see in Section 5).

4.5. ENTRY POINTS FILE [PRM\PRMENTRYPOINTS.TEXT]

This file is output by the Path Map preprocessor. It lists entry points with their position, normal and generation probability, then the number of entry points per cluster. This information is required to compute the PRM weighting for the final rendering using the PRM textures.

4.6. PRM TEXTURES [PRM*DDS]

The Path Map preprocessor outputs the textures required for indirect illumination rendering to 16bit depth RGBA DDS format. Every file name is composed from the 'pathmapfile' stub given in the level file for the entity, suffixed by an underscore and the number of the subentitity. Using the Maya exporter the entity name will be identical to the file name, and this is always recommended.

5. MAYA SCENE EXPORTER

We have developed a MEL script that can export a Maya scene into our *.level* file format. This script assumes that you have installed the Ogre3D Maya exporter plugin, as it saves the objects in Ogre3D *.xml* file format. For further information about installing and using the Ogre3D Maya exporter plugin please refer to the Ogre3D documentation.

The name of the script file is:

"GTPSceneExport.mel"

This Maya script file should be copied into Maya's default script folder:

"My Documents\maya\7.0\scripts\"

The following line should be added to *userSetup.mel* located in Maya7.0 script folder:

source GTPSceneExport;

After executing Maya a new menu should appear named *"GTP"* (if the Ogre3D plugin is installed properly the *"Ogre"* menu should also be present):

Figure 22. Maya menu bar with Ogre and GTP menus.

If we have a scene ready to export we can use the "*GTP"* menu's *"Export Scene"* entry:

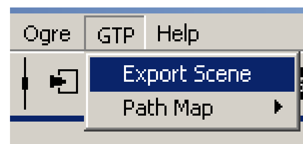

Figure 23. Export scene menu option.

This will show the *"GTP Scene Exporter"* window:

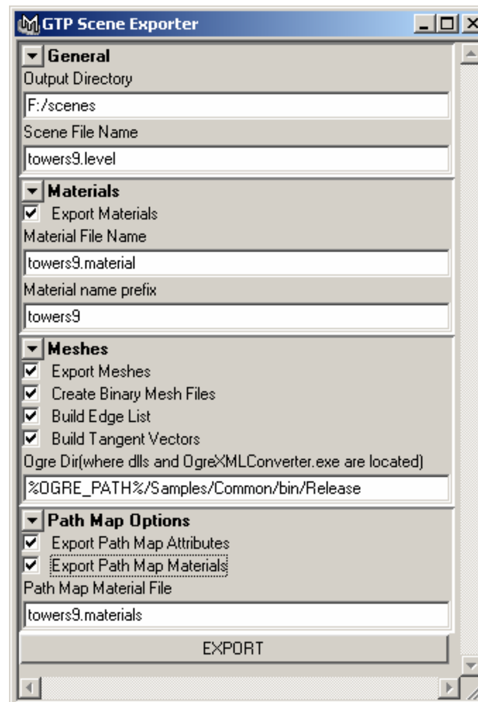

Figure 24. The GTP scene exporter dialog.

The controls have the following meanings:

General Controls:

- **Output Directory:** The directory where all output files will be created. Default: the Maya scene file directory.
- **Scene File Name:** The name of the level file to export. Default: Maya file name with *.level* extension

MaterialControls:

- **Export Materials:** If turned on the materials used by the object will be saved in an Ogre3D material file. Same as turning on the *"Export materials to Ogre .material file"* checkbox in the Ogre3D exporter. Default: off
- **Material File Name:** The name of the Ogre3D *.material* file to export. Only active if *"Export Materials"* is turned on. Default: Maya scene file name with *.material* extension.
- **Material name prefix:** The material exporter exports the materials with a name identical to their names used in Maya. This prefix will be added to the material name so you can avoid name duplications among different levels. Only active if *"Export Materials"* is turned on. Default: Maya scene file name.

Mesh Controls:

- **Export Meshes:** If turned on all shapes will be exported to an Ogre3D *.xml* file. The file names will be the name of the shape used in Maya. Note that the shapes are exported and not the transforms, so if two objects share the same shape in the Maya scene (e.g. they were created with an instance duplication) only one shape will be exported. The exported data are in object space, all transformations are saved in the *.level* file. The exported mesh data will contain vertex positions, normals and texture coordinates. Default: on.

- **Create Binary Mesh Files:** If turned on binary Ogre3D *.mesh* files will be created from the *.xml* files. Only active if *"Export Meshes"* is turned on. Default: on.

- **Build Edge List:** If turned on edge information for shadow volume computation will be stored in the binary *.mesh* files. Only active if *"Create Binary Mesh Files"* is turned on. Default: off

- **Build Tangent Vectors:** If turned on tangent vectors (for normal mapping) will be computed and stored in the binary *.mesh* file. Only active if *"Create Binary Mesh Files"* is turned on. Default: off.

- **Ogre Dir:** The full path where the Ogre3D dlls and the OgreXMLConverter.exe are located. These files are essential for binary mesh creation. Only active if *"Create Binary Mesh Files"* is turned on. Default: *"%OGRE_PATH%/Samples/Common/bin/Release"* (where OGRE_PATH is a system variable defining the Ogre3D root directory, this variable should be set if the Ogre3D version of illumination modules is used)

See Path Map Options in section 6.3.

6. LIGHT PATH MAPS PREPROCESSOR

The *Path Map* preprocessor implements the *Light Path Map* approach for indirect lighting. Its purpose is to add realism to computer games by computing dynamically changing indirect illumination. It is an improvement over ambient lighting or static light maps. Global illumination computations are performed in a preprocessing step, using ray casting and indirect photon mapping (the virtual light sources method). The contributions of virtual light samples are computed on the *GPU*, with depth mapping. Instead of computing a single light map, multiple texture atlases (constituting the *PRM*) are generated for the scene objects, all corresponding to a cluster of indirect lighting samples. Then these atlases are combined according to actual lighting conditions. Weighting factors depend on how much light actually arrives at the sample points used for *PRM* generation. This computation is also performed in a *GPU* pass.

The final result will be a plausible rendering of indirect illumination. Indirect shadows and color bleeding effects will appear. Indirect illumination will change as the light sources move. However, this comes at the price of fetching data from all *PRM* texture panes (clusters of indirect illumination) instead of just fetching a light map color. This limits the number of panes we can use. The accuracy will depend on a number of factors:

- The number of samples (entry points). Increasing this number will make preprocessing longer, but not influence rendering times. This is a command line parameter of the preprocessor.
- The number of independent entry point clusters. This is a command line parameter (see section 6.1) of the preprocessor. The number should be larger for larger scenes, and a higher number of clusters generally allows for more accuracy. However, a single piece of surface will not use all of these clusters, but only the most relevant few. If these represent too little a fraction of the actual significant clusters, undesirable edges may appear between surface elements using different clusters. A low number of overall clusters will produce nice indirect illumination, although with less fidelity to the changes of actual lighting.
- The number of entry point clusters taken into account when rendering an object. This value can be different for every object, even for those which share the same base mesh. It can be specified in Maya, and it is exported to the level file. This number should be comparable to the number of overall clusters insofar as the area from where indirect lighting may reach the surface is comparable to the complete level. A reasonable maximum is 32, more than this will not be processed by the visualization built in the preprocessor.
- The resolution of the *PRM* atlas. This value can be different for every mesh, but it is the same for all objects sharing that mesh. The resolution has to be high enough to accommodate for a UV atlas of the mesh, but, as indirect illumination tends to be low-frequency, even 16x16 or 32x32 might be a sufficient size.
- How many parts to segment a mesh into. As a mesh might be huge, it is possible that different parts receive indirect illumination from very different entry point clusters. If the mesh is treated as one, only a limited number of entry points can be considered, leaving the object dark in all but very specific lighting scenarios. Therefore, the preprocessor segments meshes into the number of sub-meshes given in Maya, exported to the level file. Individual segments can have different relevant clusters, so every part of the object will have indirect illumination.

More segments always mean more accuracy, but every segment will have its own PRM texture, with the resolution specified for the object. With more segments, the resolution should be lowered. However, as there are technical limits to how small a UVAtlas can be, heavy segmentation cannot be achieved without increasing preprocessing time.

The program performs three tasks: segment the meshes, compute the *PRM*, and use it to display the scene with indirect illumination. The *PRM*, along with the location of the sample points, is saved to files, and does not need to be computed every time. The final rendering is also implemented in the Ogre3D engine, where it can be combined with other effects. The final rendering in the preprocessor can be considered a preview.

The program is a standalone DirectX 9.0 application, compiles under Visual C++ 2003 with DirectX SDK December 2006. The source code and the precompiled *pathmap.exe* can be found in the GTP repository in the following path:

/gametools/gtp/trunk/App/Demos/Illum/pathmap/

6.1. RUNNING THE PATH MAP PREPROCESSOR

The program can be controlled by the following command line options:

```
pathmap.exe [s|p/v]     -D <media input directory>
                        -L <level file>
                        -M <materials file>
                        -O <mesh output dir>
                        -P <prm out put dir>
                        -E <number of entry points>
                        -C <number of clusters>
                        -S <shadow map resolution>
                        -U
```

Operations

s - *Segment input meshes: creates new versions of input meshes, and outputs them to the <mesh output dir> . These processed meshes will contain smaller submeshes, to which PRM textures will be assigned.*

p - *Precompute PRM textures. Outputs textures (hdr format), a new level file (with cluster assigment info), and an entry point file.*

sp - *Do both of the above.*

v - *Do neither of the above, only load and visualize results.*

Switches

D - *directory for input files (level file, material file, meshes in .x or Ogre3D xml format, texture images)*

L - *file name of level file (txt file listing meshes and entities)*

M - *file name for material file (txt file listing material name - color texture pairs)*

O - *output directory for segmented meshes and the new level file (this is also the input directory for the same files if operation 's' is not specified)*

P - *output directory for PRM textures and the entry point file. (this is also the input directory for the same if operation 'p' is not specified)*

E - *number of entry points. Must be equal to 4096×n.*
C - *number of entry point clusters overall. How many of them influences a single entity is given in the level file.*
S - *depth map resolution used for indirect illumination precomputation.*
U - *specifies entry point sampling proportional to surface area. By default, all objects receive the same number of entry points (to have more on more complex ones).*

Level files and material files can be generated with the *"GTP Scene Exporter"* MEL script provided with the illumination modules. For information about exporting objects into *.level* files see Maya export scripts in Section 75.

6.2. CONTROLS USED IN PATH MAP PREVIEW RENDERING

After preprocessing, or if no preprocessing tasks are specified, the Path Map preprocessor will visualize the scene with or without Path Map indirect illumination. This provides a way to verify results and compare them to classic techniques. A single spot light source is used for illumination, and a simple hard shadow map is used for direct shadows. Note that the final PRM weighting can be computed for more complex lighting and any shadow algorithm.

Both the camera and the light source can be moved according to the solution standard in DirectX applications: W, A, S, D keyboard buttons and the mouse. Whether the camera or the light source is moved is specified by the on-screen checkbox 'Move light'. It is possible to render the screen from the viewpoint of the camera, by checking 'Look from light'. It is easier to navigate the light while using its viewpoint. 'Turbo' will change the speed of movement for both the camera and the light source. Checking 'Entry points' will show the sample points on the surface of the scene objects, color coded according to the cluster they belong to. 'Cruise' will send the light moving on a preprogrammed path, with is only meaningful for the default space station scene.

The 'Tone scale' slider sets the lighting intensity. It scales both direct and indirect illumination, be it ambient or PRM. The 'Lighting mode' slider can be used to choose from four lighting modes: direct illumination only, direct illumination with an ambient term, direct illumination with PRM, and PRM indirect contribution only. The 'Torch distance' slider allows adding some offset to the render camera when looking from the light. If 'Move light' is also set, this creates torch-like lighting, with some shadows visible, as opposed to the basic headlight setup, where you cannot see shadowed areas.

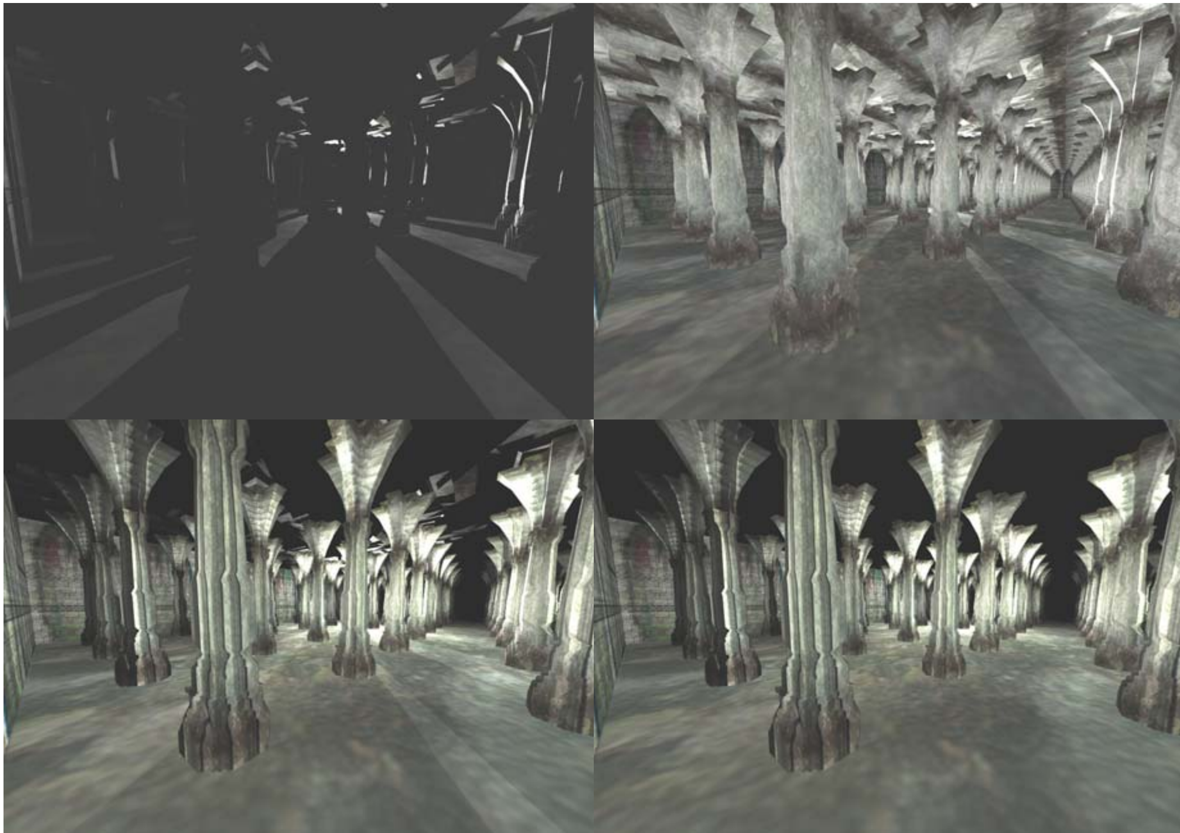

Figure 25. The dwarf chamber with direct, direct plus ambient, direct plus PRM indirect, and the PRM contribution only. The classic ambient solution is insensitive to the actual position of the light and the direction of the indirect illumination. PRM provides a more distinction and realism.

6.3. PATH MAP ATTRIBUTES IN MAYA

The path map preprocessor requires some addition information about the meshes, which are not provided in the *Ogre3D .xml* format and they don't have a representation in Maya either. These attributes can be set with a command located in the *GTP* menu and they will be saved in the *.level* file:

Figure 26. Setting Path Map attributes in Maya..

The *"Add Path Map attributes"* command adds these attributes to the selected objects (they appear under *"Extra Attributes"*).

For shapes two extra attributes are added:

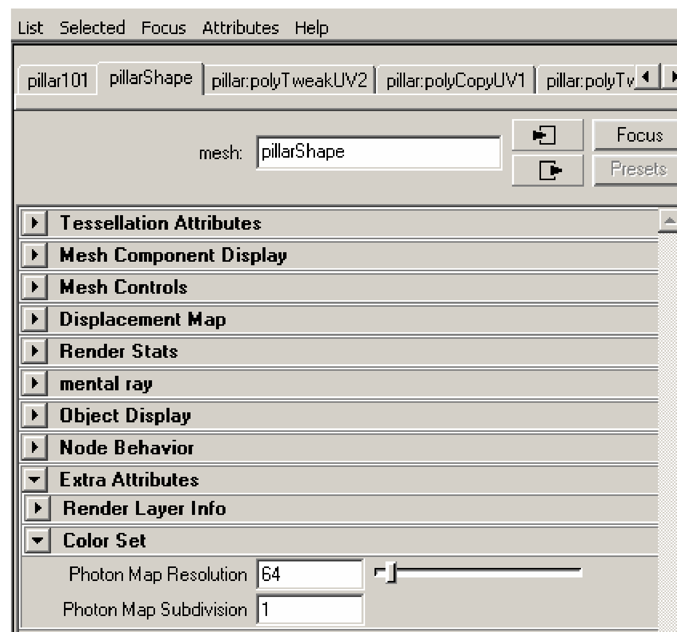

Figure 27. Shape attributes governing Path Map generation.

- Photon Map Resolution: *The path map resolution for this geometry*.
- Photon Map Subdivision: *The number of segments this geometry should be divided into.*

For transforms one extra attribute is added:

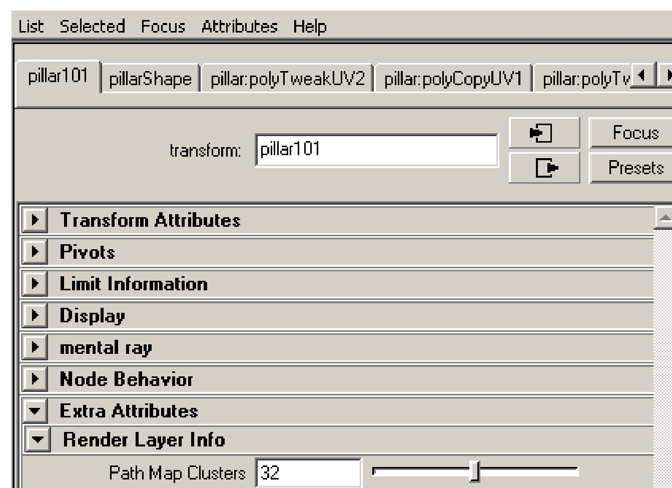

Figure 28. Transform attributes governing the number of relevant clusters to an object.

- Path Map Clusters: *The number of clusters that are relevant to this object.*

These attributes will be saved in the *.level* file after exporting the scene using the *Export Scene* command in the *GTP* menu.

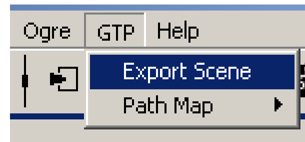

Figure 29. Exporting a scene from Maya.

This will show the *"GTP Scene Exporter"* window:

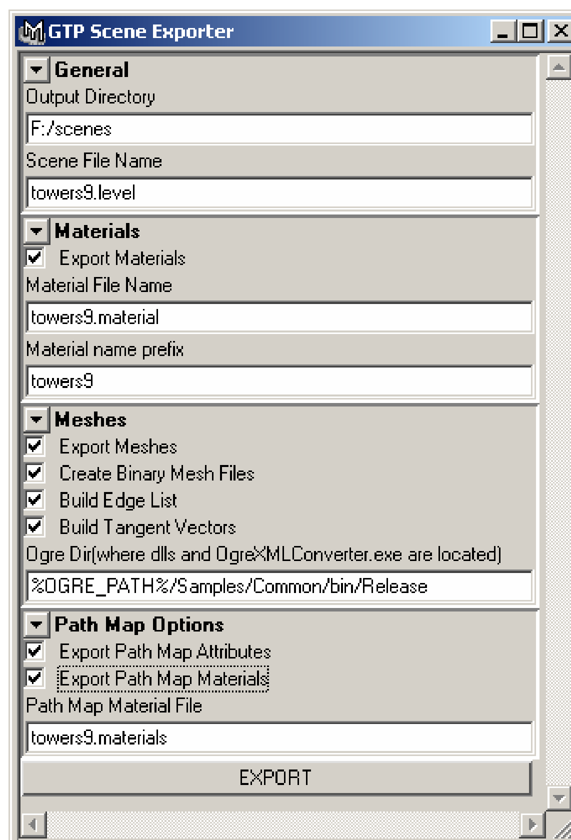

Figure 30. The GTP scene exporter dialog.

For information about the options not related to path map see Exporting Scenes from Maya.

Path Map Options:

- **Export Path Map Attributes:** If turned on, path map attributes will be saved in the *.level* file. Default: off.
- **Export Path Map Materials:** If turned on all materials present in the scene will be saved to a format that can be used by *pathmap.exe*. These materials will use the first texture channel of the Maya material -or it's diffuse color if no textures are assigned- to display objects. Only active if *"Export Path Map Attributes"* is turned on. Default: off.
- **Path Map Material File:** The name of the material file where materials for *pathmap.exe* will be saved. Only active if *"Export Path Map Materials"* is turned on. Default: Maya scene file name with *.materials* extension

7. OBSCURANCES PREPROCESSOR

The obscurances preprocessor is an off-line program to generate lightmaps describing the indirect illumination using the obscurances algorithm. The obscurance algorithm approximates radiosity, but involves a much lower computational cost. Its main advantage lies in the fact that this technique considers only neighboring interactions instead of attempting to solve all the global ones. Another advantage of this algorithm is that it is decoupled from direct illumination computation, thus it bakes only indirect illumination into textures while allows completely dynamic lighting when direct reflections are computed. The obscurances algorithm can deal with any number of moving light sources and generates color bleeding as well.

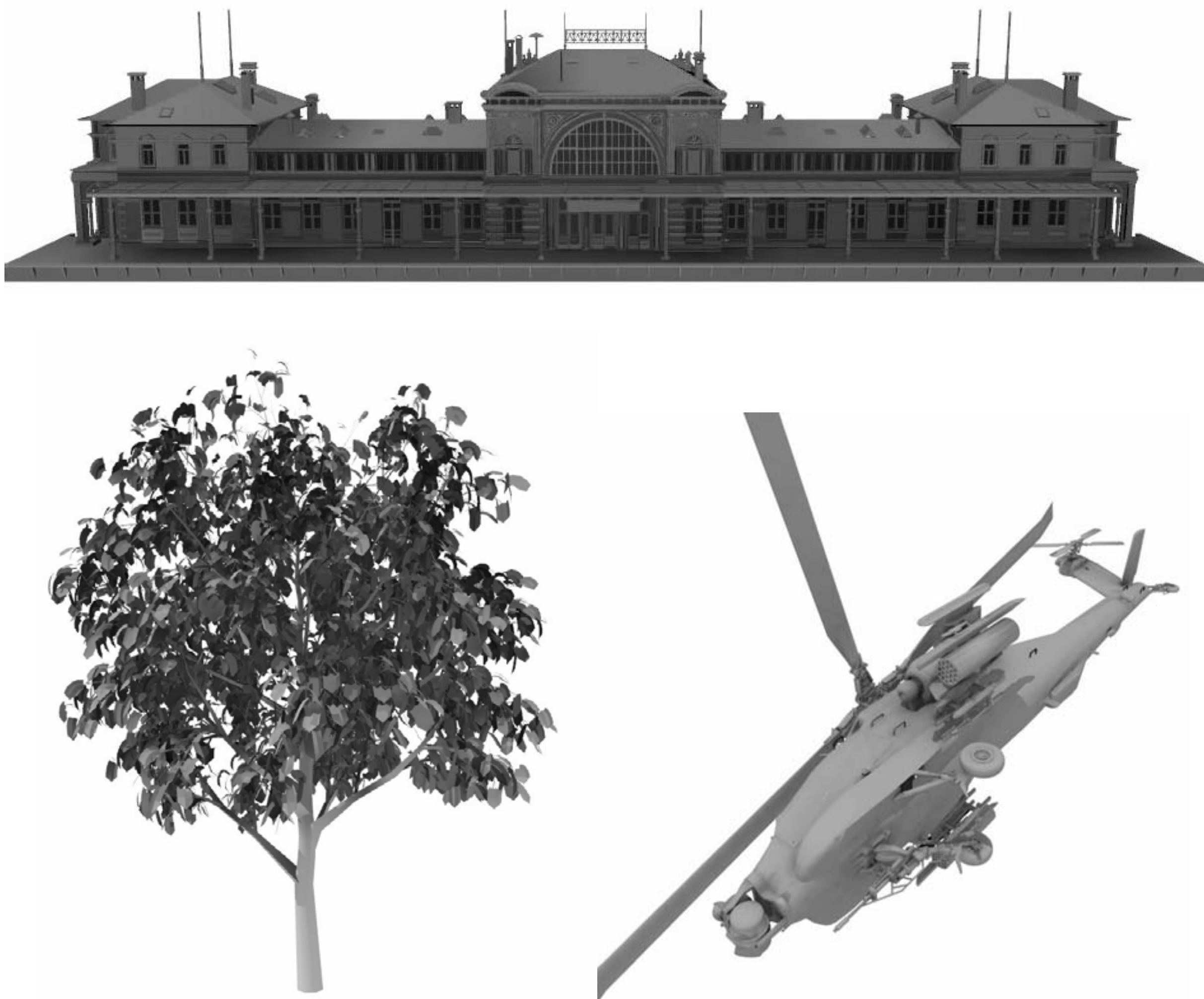

Figure 31. Final results generated by the obscurances method.

The obscurance preprocessor can be found in the *Obscurances* directory of the repository.

7.1. IMPLEMENTATION OF THE OBSCURANCES PREPROCESSOR

The obscurances preprocessor is a standalone and multiplatform application that allows us to load Ogre3D mesh and XML files and generates an image file containing the obscurance information. This image file can be applied to Ogre scenes as a lightmap. All the libraries used in this application are multiplatform, the GUI is made using wxWidgets library, for the graphical stuff we use the OpenGL API and GLSL and for the XML parsing process we use the TinyXML library.

The most important classes of the application are `vcObscuranceMap` and `CMesh`. The first contains all the necessary information and functionality to compute the obscurance map and has a method that returns a pointer to the resulting image (`vcObscurerGenerateImage`). `CMesh` contains the geometry of the scene and carries the scene parsing plus the hardware-accelerated structures used in the process to generate the obscurance map.

Some of the parameters that can affect the performance of the application and the image quality of the result are already adjusted. The number of directions taken is set to 180 (3 steps × 60 directions / step), but this number can be modified changing the step and iteration variables in the code. Changing steps is not a problem. The greater the number, the better the result, but it takes longer to finish. Care should be taken when the iterations per step is increased since high values can result in some image glitches due to saturation of the `fp16` values used for accumulative blending.

Another variable that can be changed is d_{max} that represents the maximum distance where light interactions are computed. The default is 0.3. The variable can be found in the `RenderTransfer` function.

Another parameter that can be tweaked is the relative resolution between the projection planes used to calculate the obscurance transfer and the resolution of the resulting obscurance map. This can be done in `vcObscurerGenerateImage` at the definition of `ResX` and `ResY`. Now they are set to 2.0 so the projection images have double the resolution of the desired obscurance map. This can be changed to 4.0 or 8.0 but then the number of iteration should be decreased to avoid the saturation of `fp16` buffer used for accumulation.

Finally, there can be some problems while filtering the obscurance map. There are three ways to solve that:

- The first method is to expand the charts of the resulting obscurance map.
- In the first method the obscurance map created by the application is normalized (RGB components are divided by A component). If the application does not normalize components and the normalization is done on the GPU just before rendering the model with obscurances, filtering problems will be avoided and chart expansion will not be necessary anymore.
- A third solution can be not using the hardware filtering and program the filtering in the shader. This filtering shader should discard all the samples with zero alpha component.

7.2. RUNNING THE OBSCURANCES PREPROCESSOR

The GPUObscurances Generator Program has a very simple GUI. This graphical interface guides the user through four steps in order to set up the parameters needed to calculate the lightmap.

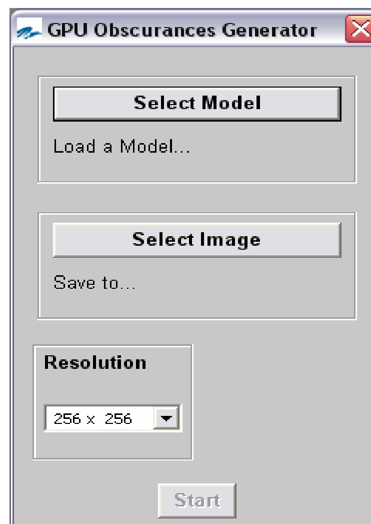

Figure 32. GUI of the obscurances preprocessor.

In order to generate the obscurance map, the user must perform the following steps:

- The first step is selecting the mesh. The mesh must be in Ogre3D XML format. After selecting the model, its data will be parsed by the TinyXML library and stored in the application internal storage structure.

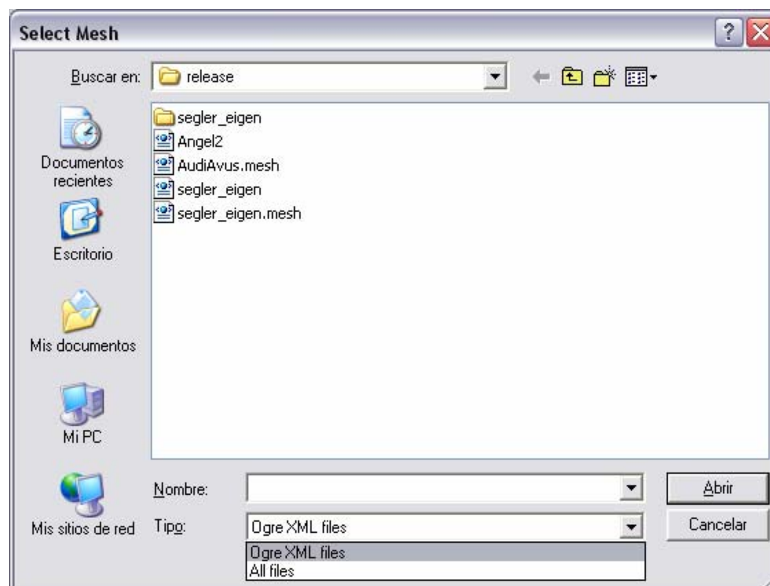

Figure 33. Model Selection.

- In the second step the obscurance map file should be specified. The generated image will be in *.bmp format.

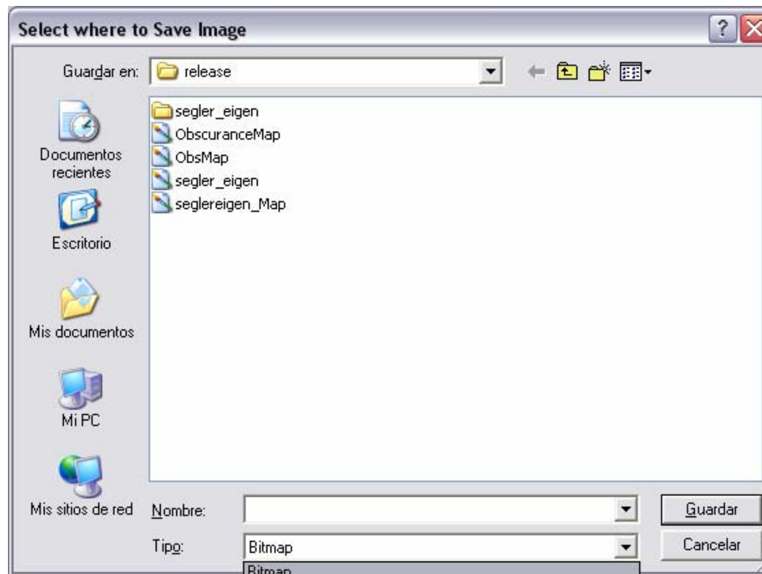

Figure 34. Obscurance map saving.

To store the obscurance map in bitmap format we also use the wxWidgets library.

- The third step lets the user choose the desired image resolution.

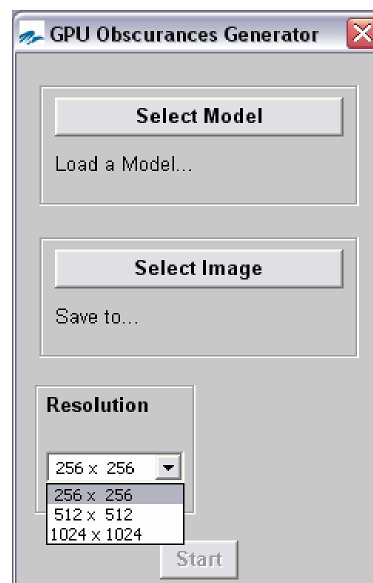

Figure 35. Selecting Obscurance map resolution.

- Finally, pushing the start button, a progress bar will appear while the obscurance generation process is being performed.

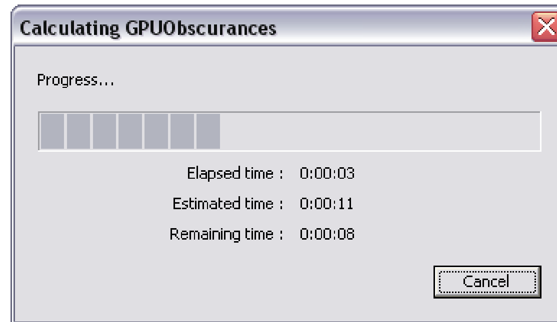

Figure 36. Obscurances generation progress bar.

After completion of the progress bar, the obscurance map is stored as a *.bmp file in the destination folder chosen in step 2.

8. BILLBOARD TREE PREPROCESSOR

The *Billboard Tree Preprocessor* provides a set of tools for creating and previewing 3D tree. The generated tree is represented by a set of billboards, called *billboard cloud*. The billboards are built automatically by a clustering algorithm. Unlike classical billboards, the billboards of a billboard cloud are not rotated when the camera moves, thus the expected occlusion and parallax effects are provided. On the other hand, this approach allows the replacement of a large number of leaves by a single semi-transparent quadrilateral, which considerably improves the rendering performance. A billboard cloud well represents the tree from any direction and provides accurate depth values, thus the method is also good for shadow algorithms. The billboard cloud decomposes the original object into subsets of patches and replaces each subset by a billboard. These billboards are fixed and the final image is the composition of their images.

The *Billboard Tree Preprocessor* have been developed using the Ogre3D engine, it is compatible with both OpenGL and Direct3D, and requires the following dependencies.

- Ogre3D
- Boost
- FCollada
- ImageDebugger

These dependencies can be found in the following repository folder:

www.gametools.org/repos/gametools/NonGTP/

In the figure below the complete production pipeline is shown.

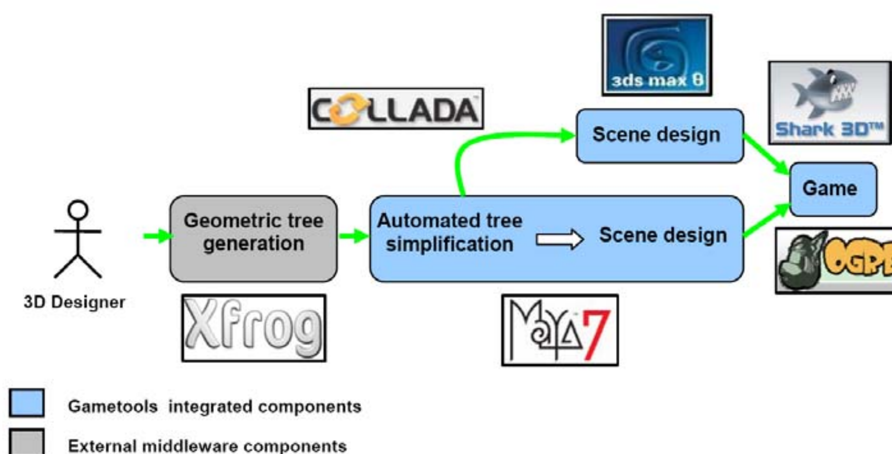

Figure 37. Billboard tree generation and production pipeline.

The main documentation is divided into two manuals in the following folders.

The user manual covers all the features available in this preprocessor:

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/OGRE/doc/userManual/

The developer manual it is basically a Doxygen documentation of the main components of this preprocessor. This documentation is useful if the user wants to extend the application in some specific way:

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/OGRE/doc/devManual/

All the media files used during the billboard generation can be found at:

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/OGRE/media

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/OGRE/media

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/OGRE/media

The media folder is the default folder where you will find the textures, meshes, and shaders used by this preprocessor.

- The default shaders are placed in media/general.
- There is a sample tree in media/chestnut and a sample configuration file in /media/chestnut/leaves that shows how to process the sample tree.

Project make files folder:

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/OGRE/scripts/IBRBillboardCloudTreeGenerator.sln

Source code folders:

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/OGRE/include/

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/OGRE/src/

8.1. RUNNING THE BILLBOARD TREE GENERATOR

The process of creating a billboard cloud tree consists of the following steps:

1. The 3D content creator creates a high detailed polygonal tree with some available tools, such as *Xfrog* or *PovTree*.
2. The high detailed tree model should be decomposed into the leaves and the trunk.
3. The leaves model is converted to the Ogre3D Mesh file format. We have provided exporters available for Maya or 3ds Max for this conversion.

   The trunk model is not processed with the generator tool and the designer must create a set of detail levels of the original mesh using a simplification software, for example, the tools of the geometry WP. The *Billboard Tree Preprocessor* receives as input the leaves and the texture used to map each leaf.

The output generated is a billboard cloud mesh and three texture atlases as described here:

- The billboard cloud model that replaces the original leaves model.
- The texture atlas that contains the colors of all leaves. This texture will be used by lower-end graphics hardware with standard texture mapping.

- The rotated leaf texture atlas contains the leaf placed with different orientations.
- The leaves distribution atlas texture. This texture will be used with the rotated leaf texture atlas in the technique called *indirect texturing* .

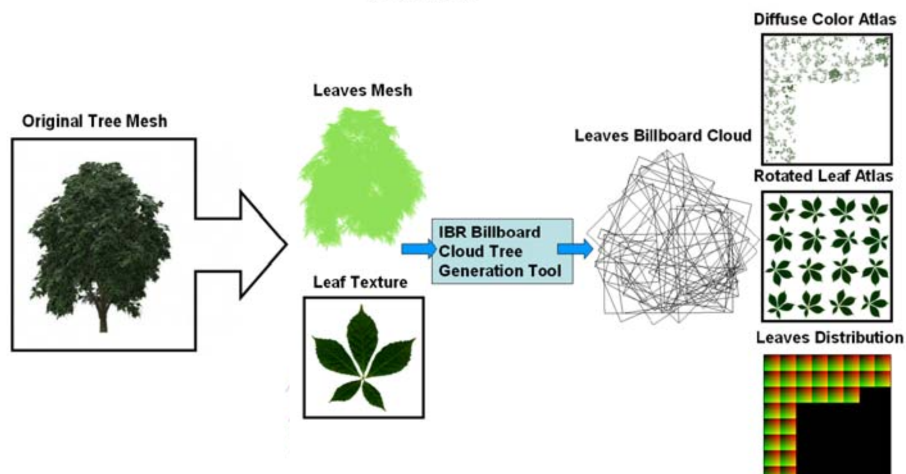

Figure 38. Sample polygonal tree decomposed in two models. The leaves model will be processed with the Billboard Tree Preprocessor.

The *Billboard Tree Preprocessor* is a command line application managed through configuration files. For each kind of tree, we should create a configuration file. The application must have as input parameter the configuration file as shown here:

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/OGRE/bin/Release

IBRBillboardCloudTreeGeneratorCmd.exe −cfg . . / . . /media/chestnut/leaves/sample.cfg

Configuration file description:

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/OGRE/media/chestnut/leaves/sample.cfg

The configuration file contains all the parameters needed to obtain the desired output, including

- The media folder parameters.
- The input file parameters.
- The output file parameters.
- The user custom parameters.
- Input folders.
- Temporary folders.
- Output folders.
- Rotated leaf texture atlas parameters:

```
# This is the sample input leaf texture filename
Diffuse Color Entity Texture Name chestnutLeaf.png
# The output texture atlas file name
Diffuse Color Entity Texture Atlas Name chestnutRotatedLeafAtlas.png
# The output texture atlas bit range (8 bits , 16 bits , 32 bits per channel)
Diffuse Color Entity Texture Atlas Bit Range 8
# The output texture atlas size in pixels
Diffuse Color Entity Texture Atlas Size 512
# The number of  textures with different leaf orientations we want
Diffuse Color Entity Texture Atlas NumSamples 16
```

- Diffuse color texture atlas parameters:

```
# The output texture atlas file name
Diffuse Color Texture Atlas Name diffuseColorAtlas.png
# The output texture atlas bit range (8 bits , 16 bits , 32 bits per channel)
Diffuse Color Texture Atlas Bit Range 8
# The output texture atlas size in pixels
Diffuse Color Texture Atlas Size 1024
# The size that must have each billboard texture in the texture atlas
Diffuse Color Texture Size 16
```

- Leaves distribution texture atlas parameters:

```
# The output texture atlas file name
Indirect Texture Atlas Name indirectTextureAtlas.png
# The output texture atlas bit range (8 bits , 16 bits , 32 bits per channel)
Indirect Texture Atlas Bit Range 8
# The output texture atlas size in pixels
Indirect Texture Atlas Size 1024
# The size for each billboard texture in the texture atlas
Indirect Texture Size 16
```

- User custom parameters:

```
# This is the maximum number of billboards desired
EntityClusters MaxClusters 1024
```

Note that the *Billboard Tree Preprocessor* has more custom parameters. Refer to the user manual included with the tool.

The *Billboard Tree Preprocessor* has three different view modes:

- *Diffuse Color Texture Atlas View Mode*: This view mode shows the diffuse color texture atlas generation process, and each billboard plane with its texture mapped leaves.
- *Indirect Texture Atlas View Mode*: This view mode shows the leaves distribution impostor texture atlas generation.
- *Rotated Leaf Texture Atlas View Mode*: This view mode shows the rotated leaf texture atlas.
- *Billboard Cloud Final View Mode*: In this mode you can look at your billboard cloud mesh using indirect texturing or standard texture mapping techniques, is good to check that the resulting billboard cloud tree has been generated correctly. Here we can apply a tweak to the indirect texturing leaf position.

Views controls:

- `Key F1:` Enable Diffuse Color Texture Atlas View Mode.
- `Key F2:` Enable Indirect Texture Atlas View Mode.

• `Key F3:` Enable Rotated Leaf Texture Atlas View Mode.

• `Key F4:` Enable Billboard Cloud Final View Mode.

• `SPACE:` This key iterates along all the billboards generated to see how each group of leaves has been placed on them. This key is working only in the Diffuse Color Texture Atlas View Mode and Indirect Texture Atlas View Mode.

Navigation controls:

These controls are enabled only in the Billboard Cloud Final View Mode. With them you can go around the billboard cloud.

- `Key E/Key D:` Accelerate/Brake
- `Key S/Key F:` Turn
- `Key PGUP/Key PGDOWN:` Shift

8.2. BILLBOARD TREE PLUGIN IN MAYA

The billboard tree preprocessor has been integrated into Maya. The *Billboard Tree Maya* plug-in can be found at:

/gametools/gtp/trunk/Lib/Illum/IBRBillboardCloudTrees/Maya

The *Billboard Tree Plugin* have been developed for improving the integration of the billboard cloud generation process with Maya Unlimited 5.0, 6.0, and 7.0.

Before starting to work with the plug-in, we should check if the Billboard Cloud Generator Plug-in is loaded. For checking this, we can click on the Windows-Settings/Preferences-Plug-in Manager. . . . If we cannot find the plug-in called BillboardCloudGenerator.mll, then we should load it executing the following steps:

1. Click on the Browse button, search the plug-in package folder bin/release/.

2. We find the file called BillboardCloudGenerator.mll and open it.

3. Now this plug-in will appear in the Maya plug-in list.

We should check if the loaded checkbox of the plug-in is marked. When the plug-in is loaded, another Maya window is created that logs all steps executed by the plug-in. This output window is very useful to check if there have been errors caused by the plug-in.

Figure 39. Main icons of the Billboard Tree Plugin.

```
Creating resource group General
Registering ResourceManager for type Material
Registering ResourceManager for type Mesh
Registering ResourceManager for type Skeleton
Loading library OgrePlatform.dll
OverlayElementFactory for type Panel registered.
OverlayElementFactory for type BorderPanel registered.
OverlayElementFactory for type TextArea registered.
Registering ResourceManager for type Font
ArchiveFactory for archive type FileSystem registered.
ArchiveFactory for archive type Zip registered.
DevIL version: Developer's Image Library (DevIL) 1.6.7 Oct 28 2005
DevIL image formats: bmp dib cut dcx dds gif hdr ico cur jpg jpe jpeg
Registering ResourceManager for type HighLevelGpuProgram
*-*-* OGRE Initialising
*-*-* Version 1.0.6 (Azathoth)
Added resource location '../../../media/chestnut' of type 'FileSystem
Added resource location '../../../media/maple' of type 'FileSystem' t
Parsing scripts for resource group General
Parsing script chestnutClusterizedEntities.material
Finished parsing scripts for resource group General
```

Figure 40. Output log Billboard Cloud Plug-in window.

The steps necessary to use the *Billboard Tree Plugin* are also shown in the billboard tree generation video tutorial available in the repository:

www.gametools.org/projects_udg/gt_ibr/videos/gtIBRGeneration.wmv

8.3. MODELLING BILLBOARD CLOUD TREES IN 3DS MAX

The Billboard Tree Preprocessor is able to export the foliage billboards in a 3D interchange format called *Collada*, which enables us to use the simplified trees in several professional tools, such as 3ds Max or Maya Unlimited. The following folder contains a .max sample file with sample billboard cloud trees with all the rendering properties detailed in this section. Those files can be exported to Ogre3D or other game engines, such as Shark3D.

/gametools/gtp/trunk/App/Demos/Illum/IBRBillboardCloudTrees/3dsMax/indirectTexturing/

We have developed an indirect texturing 3ds max *Effect File* to allow interactive previsualizations of the game scenes inside 3ds max with the indirect texturing technique. This will allow designers to control the final scene appearance in the scene composition process. The indirect texturing *Effect File* contains the shading properties to be used to render the billboard trees inside 3ds max.

When the designer loads the *Collada* files of the billboard trees generated by the *Billboard Tree Preprocessor,* they will appear as shown below without the final shading properties.

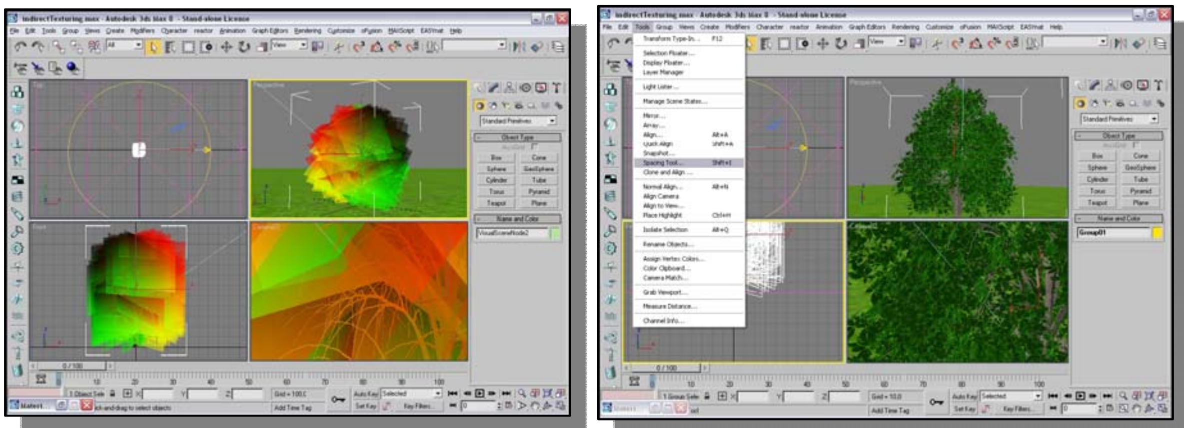

Figure 41. Left: Billboard tree without the indirect texturing technique being used. Right: Billboard tree using the indirect texturing Effect File.

In order to use the indirect texturing *Effect File* to enable the expected foliage appearance the user should activate the material window (*Key M*), open the *Material/Map Browser* and choose *DirectX 9 Shader.* Then the user has to load the following indirect texturing *Effect File*:

/gametools/GTP/trunk/App/Demos/Illum/IBRBillboardCloudTrees/3dsMax/indirectTexturing/indirectTexturingLighting.fx

The billboard cloud meshes will appear shaded with the foliage textures using the indirect texturing technique. After scene composition, the forest scenes can be exported to the game engines such as Ogre3D using the respective game scene exporters.