User guide: CloudyDay - Rendering of clouds, atmosphere and light shafts in HDR for testing Computer Vision algorithms

Michael Beham, Vienna University of Technology and AIT - Austrian Institute of Technology

Figure 1: ***CloudyDay*****: Rendering of 3D clouds (stratus, cumulus clouds at center), 2D clouds (stratocumulus clouds at top) atmosphere, terrain, and airplanes to test Computer Vision (CV) algorithms.**

Abstract – Rendering of clouds, atmosphere, and other natural phenomenon is an important topic in computer graphics. In this technical report, we present a novel solution, which uses different techniques to generate a realistic representation of the sky. We present a billboard-based approach to create clouds. We use half-angle slicing to generate volumetric shadows. The resulting shadow map is then used for casting shadows on the terrain, the clouds, and other objects. We also use and compare different atmosphere models and providing light shafts. Furthermore, *CloudyDay* provides HDR mapping, a bloom effect, colour grading as well as some natural phenomenon like rain.

We develop *CloudyDay* to test an autonomous flying robot. We present several enhancements, which consider the specific requirements of this specific application area. All objects can be created by an artist. This is great workflow, if a specific test-case should be created. However, creating a lot of different variations of an object is a time-consuming task. A more reasonable way is to create the shapes with procedural modelling. This technique enables to create objects (in this paper clouds, atmosphere,...) and vary the representation by varying the parameters.

Table of contents:

Introduction

Figure 2: Simulated sunset with terrain, airplane and lightshafts.

This report gives a guide to setup *CloudyDay*. This library gives a realistic representation of a sky. *CloudyDay* provides following features:

- Atmosphere:
  - O'Neal approach [O'N05]
  - Bruneton et al. approach [Bruneton 08]
- Objects
  - Terrain and other objects
  - Variance soft shadows
- 3D clouds
  - for low- and middle located clouds
  - Volumetric shadows ,and shadows at terrain and other objects
  - Simulation of cloud dynamics
- 2D clouds
  - for middle- and high located clouds
  - Parallax mapping
  - Displacement shader
  - Nature phenomena (rain and fog)
- Post-Processing effects
  - HDR Mapping
  - Light Shafts
  - Glare and star effect
  - Colour grading

Requirements

We create *CloudyDay* with **OpenSceneGraph 3.0**. As developing environment, we use **Microsoft Visual Studio 2010.** *CloudyDay* is tested with a **Intel i5** processor with **8 GB main memory**. To enable all features of *CloudyDay*, the graphic card has to support tesselation shaders (**OpenGL 4.0**). We test *CloudyDay* with a **Nvidia's Geforce 420 GT**, and **Geforce 570 GTX** graphic card.

Moreover, we need following libraries to compile *CloudyDay*:

- OpenGL and GLEW library
- Carve library
- TinyXML 2 library
- OpenThreads
- OpenSceneGraph 3.0 with osgUntil, osgDB, osgGA, osgText, and osgViewer

Setup CloudyDay

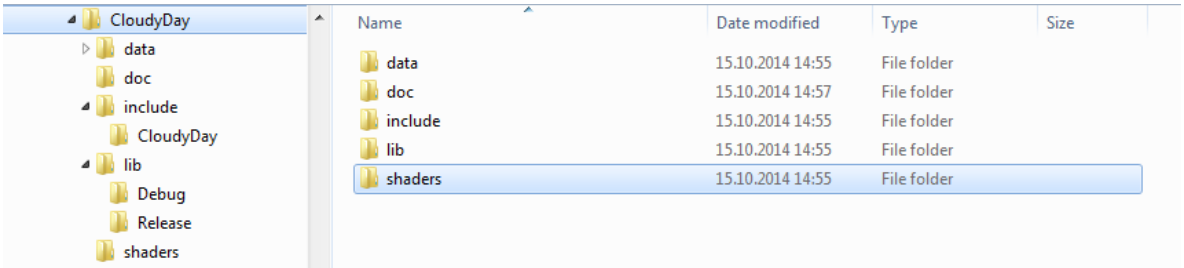


Figure 3: *CloudyDay* archive

The first step is to create a new (Visual Studio) project and copy all files in the directory. The *shader* directory has to be added. Copy it to the directory that contains the project file (*.vcxproj). Furthermore, the data directory has to be added. We copy it to the directory, which contains the *.sln file. An example is shown in figure 4.

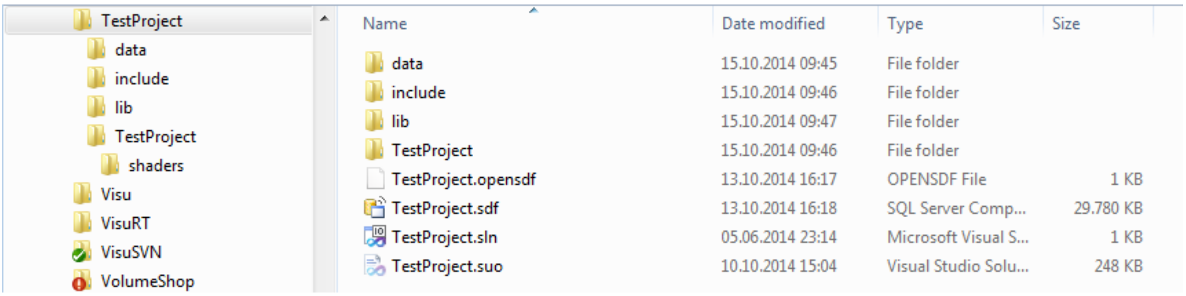


Figure 4: New project directory

The next step is to add the header files and the library. We copy all files and directories that are located in the include directory of the *CloudyDay* archive to the include directory. The CloudyDay.lib, which is located in the lib directory of the *CloudyDay* archive, have to be copied to the library directory (e.g., lib). The next step is to set the additional include directory and additional library directory in Visual Studio. We set at "Properties\\C/C++\\General\\Additional Include Directories" the path to the include directory (e.g., "$(ProjectDir)/include/CloudyDay"). We also set at "Properties\\Linker\\General\\Addidional Library Directories" the path to the lib file ("$(ProjectDir)/lib"). Then, we add the path to the CloudyDay.lib at "Properties\\Linker\\Inpud\\Additional Dependencies". Now, *CloudyDay* can be used. Note, that also the OpenSceneGraph libraries have to be added. In section: "Requriments" all libraries are listed, which are needed for *CloudyDay*. Furthermore, a step by step guide is given in the chapter Appendix A - Creating a new project.

The next chapter gives all instructions to create a new scene with *CloudyDay*. The second chapter gives all instructions to integrate parts of *CloudyDay* into an existing project.

Setup a new scene

This chapter gives instructions to configure *CloudyDay*. The resulting scene contains a terrain, different types of clouds (3D and 2D), rain, some post-processing effects as well as shadows.

Configuration of terrain and atmosphere model

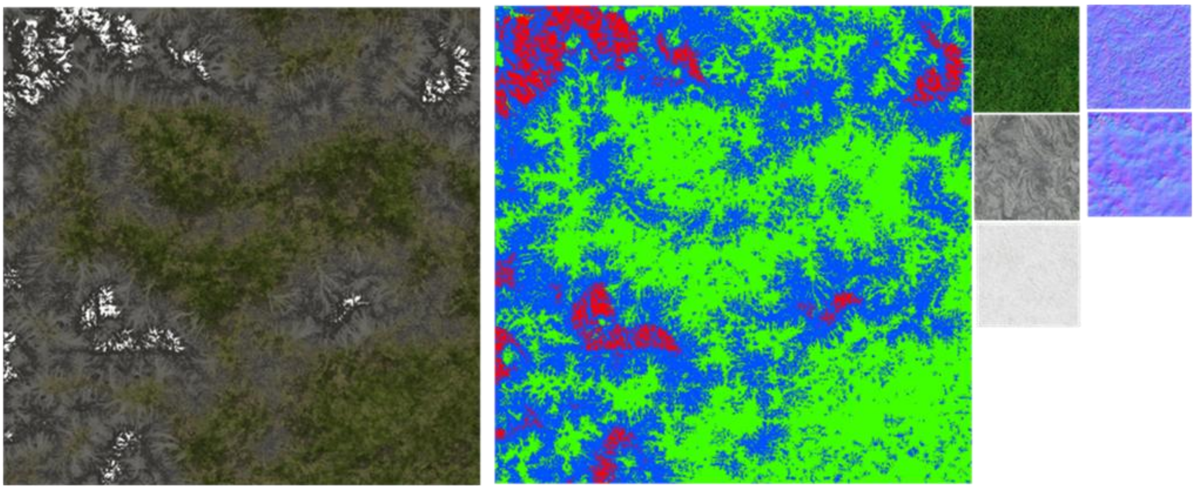


Figure 5: Textures to generate a terrain.

The first step is to configure the terrain and atmosphere. We configure both as follows:

```
osgCloudyDay::TerrainConfig* terrain_config = new osgCloudyDay::TerrainConfig();
terrain_config->SetTesselationShader(false);

terrain_config->SetPath2File("mountains.obj");
terrain_config->SetPath2DiffuseTexture("mountain_diffuse.bmp");
terrain_config->SetPath2DefinationTexture("definition1.tga");
terrain_config->SetPath2HeightTexture("mountain_heightmap.tga");

terrain_config->AddPath2Texture("snow.bmp");
terrain_config->AddPath2Texture("terrain_rock4.bmp");
terrain_config->AddPath2Texture("terrain_grass.bmp");
terrain_config->AddPath2Texture("boulder.bmp");
terrain_config->AddPath2NormalTexture("rock_bump6.tga");
terrain_config->AddPath2NormalTexture("rock_bump4.tga");

Scene::m_skydome = new osgCloudyDay::SkydomeHimmel();     //Bruneton approach
//Scene::m_skydome = new osgCloudyDay::SkydomeMie();      //Mie approach
```

First, we create a *TerrainConfig* object, and to determine, if the tesselation shader approach should be used or the standard approach. If the tesselation shader approach should be used, we set the *SetTesselationShader()* method to true. The next step is to set the paths to the 3D model and all textures. An example of the textures is shown in figure 5. Additionally, we need to create a *Skydome* configuration object. The user can select between the O'Neal [O'N05] and Bruneton et al. [Bruneton08] approach.

The last step is to pass the terrain configuration object to the constructor of the scene object. An example is given in the next line:

```
my_scene = new CloudyDay(terrain_config, false, false, false, true, true);
```

The first parameter is the terrain configuration object. The other parameters are need to set rain and the post-processing effects. We discuss the other parameters later.

Configuration of 3D clouds layers

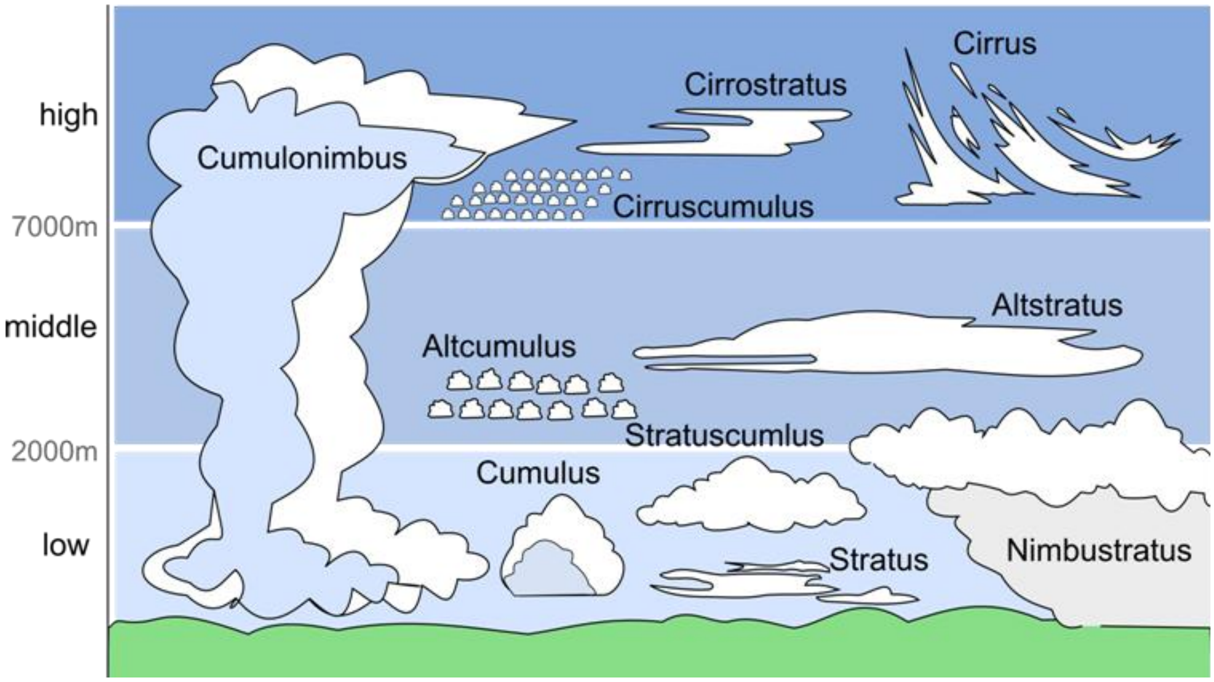


Figure 6: Overview of different cloud types.

The first step is to configure the 3D cloud layer. We use following code:

```
//Cloud Layer:
osgCloudyDay::CloudScene::GetStates()->AddLayer(0, osgCloudyDay::CloudScene::CT_Cumulus);
osgCloudyDay::CloudScene::GetStates()->setOvercast(0, 0.002551f/1.f);
osgCloudyDay::CloudScene::GetStates()->setMiddlePoint(0, osg::Vec3(0.f,0.f, 2000.f));
osgCloudyDay::CloudScene::GetStates()->setSize(0, osg::Vec3(10000.f, 10000.f, 10.f));
osgCloudyDay::CloudScene::GetStates()->setMeasureOvercast(0, 0.0f);
osgCloudyDay::CloudScene::GetStates()->setClot(0, 10);
osgCloudyDay::CloudScene::GetStates()->setVariance(0, 0.125f);
osgCloudyDay::CloudScene::GetStates()->setColour(0, osg::Vec4(1.f, 1.f, 1.f, 1.f));
```

The first step is to determine the cloud type of the 3D cloud layer. *CloudyDay* supports following cloud types (an overview of different cloud types is depicted in figure 6):

- **Cumulus:** Cumulus clouds are very „puffy" in appearance. They have a flat base, and occur at low altitude. Cumulus clouds are white [WCloud14].
- **Stratus:** These clouds are flat, hazy, and featureless. They occur at low altitude, and their colour varies between dark gray to nearly white [WCloud14]. Stratus clouds can result in rain.
- **Stratocumulus:** Similar to cumulus clouds, but large dark, rounded masses. Instead of cumulus, stratocumulus clouds usually occur in groups, lines, or waves [WCloud14]. They occur at low altitude.
- **Altocumulus:** Altocumulus clouds are puffy" in appearance. Altocumulus clouds are white. In difference to cumulus clouds, they are located at low altitude [WCloud14].
- **Altostratus:** Similar to stratus clouds, generally uniform gray to bluish-gray sheet or layer, lighter in colour than nimbostratus and darker than high cirrostratus. They are located at middle altitude [WCloud14].
- **Nimbostratus :** This cloud type has a considerably vertical and horizontal extent. They clouds distribute over a wide area. They are located in low-to-middle altitude [WCloud14]. They are associated with rain.

- **Cumulonimbus:** Cumulonimbus clouds are dense towering vertical clouds. They form alone in clusters, or along cold front squall lines. They are associated with thunderstorms and atmospheric instability [WCloud14].

After definition of the cloud layer type, the amount of overcast has to determine (between 0 and 1). We also need to define the position of the cloud, and the size of the cloud layer. The clouds of the layer are grouped to clots. So, we need to set the number of clots and the variance attribute. The variance attribute determines the size of the clots. Then, the colour of the cloud has to be determined. This parameter is needed to create greyish or white stratus clouds. *CloudyDay* supports to create some layers of clouds. E.g.: To create a cloud with cumulus and stratus clouds, we use following code:

```
//Cumulus clouds
osgCloudyDay::CloudScene::GetStates()->AddLayer(0, osgCloudyDay::CloudScene::CT_Cumulus);
osgCloudyDay::CloudScene::GetStates()->setOvercast(0, 0.002551f/1.f);
osgCloudyDay::CloudScene::GetStates()->setMiddlePoint(0, osg::Vec3(0.f,0.f, 2000.f));
osgCloudyDay::CloudScene::GetStates()->setSize(0, osg::Vec3(10000.f, 10000.f, 10.f));
osgCloudyDay::CloudScene::GetStates()->setClot(0, 10);
osgCloudyDay::CloudScene::GetStates()->setVariance(0, 0.125f);
osgCloudyDay::CloudScene::GetStates()->setColour(0, osg::Vec4(1.f, 1.f, 1.f, 1.f));

//Stratus clouds
osgCloudyDay::CloudScene::GetStates()->AddLayer(1, osgCloudyDay::CloudScene::CT_Stratus);
osgCloudyDay::CloudScene::GetStates()->setOvercast(1, 0.0011525f);
osgCloudyDay::CloudScene::GetStates()->setMiddlePoint(1, osg::Vec3(0.f, 0.f, 1000.f));
osgCloudyDay::CloudScene::GetStates()->setSize(1, osg::Vec3(20000.f, 20000.f, 10.f));
osgCloudyDay::CloudScene::GetStates()->setClot(1, 10);
osgCloudyDay::CloudScene::GetStates()->setVariance(1, 0.25);
osgCloudyDay::CloudScene::GetStates()->setColour(1, osg::Vec4(0.75f, 0.75f, 0.75f, 0.5f));
```

Configuration of particular 3D clouds:

Figure 7: Simulated cumulus clouds and cirrostratus clouds.

Besides of 3D cloud layers, *CloudyDay* supports to create particular clouds. We support following approaches:

- **Random generated cloud:** Single cloud, which is generated with a random generator

- **Wang Approach:** Generation of a cloud (layer) using a 3D modelling application. The user can define bounding box, in which *CloudyDay* distributes billboards.
- **Creation using a watertight 3D model:** Generation of a cloud using a 3D model. CloudyDay distributes the billboards in the 3D model.
- **Simulation:** Simulation of clouds using a cellular automate.

First, we need to create a configuration object as follows:

```
osgCloudyDay::CloudState* clouds = new osgCloudyDay::CloudState();
```

To add a cloud resulting from a watertight 3D model, we use following code:

```
clouds->AddCloud(osgCloudyDay::CloudScene::CT_Cumulus, "Model/standford_bunny.obj",
       osg::Vec3(0.f,0.f,2125.f), osg::Vec3(100.f, 100.f, 100.f),
       CloudState::CStG_Voxel, osg::Vec4(1.f, 1.f, 1.f, 1.f));
```

First, we need to set the cloud type. Then, we need to determine the path to the watertight 3D model. We also need to determine the position of the cloud, and the size. Furthermore, we need to set the algorithm parameter to *Voxel*. Last, we need the colour attribute.

To add a cloud resulting from a watertight 3D model, we use following code:

```
clouds->AddCloud(osgCloudyDay::CloudScene::CT_Cumulus, "",
       osg::Vec3(0.f, 0.f,2125.f), osg::Vec3(100.f, 100.f, 100.f),
       CloudState::CStG_Simulation, osg::Vec4(1.f, 1.f, 1.f, 1.f));
```

As seen in this code, we need to set the algorithm attribute to Simulation.

To add a cloud resulting from a 3D model with bounding boxes, we use following code:

```
clouds->AddCloud(osgCloudyDay::CloudScene::CT_Cumulus, "wangcloudtest.obj",
       osg::Vec3(0.f,0.f,2125.f), osg::Vec3(100.f, 100.f, 100.f), CloudState::CStG_Wang,
       osg::Vec4(1.f, 1.f, 1.f, 1.f));
```

A particular 3D cloud, created with a random generator, is generated as follows:

```
clouds->AddCloud(osgCloudyDay::CloudScene::CT_Cumulus, osg::Vec3(0.f,   0.f, 1000.f),
       osg::Vec3(100.f, 100.f, 100.f), osg::Vec4(1.f, 1.f, 1.f, 1.f)));
```

Configuration of 2D clouds:

Figure 8: Cirrus cumulous clouds in the background.

In the previous section we discuss clouds, which are located at low and middle altitude. The literature also differs between the following high-altitude clouds:

- **Cirrus clouds:** Cirrus clouds appear in thin, and wispy strands. They are white or light gray in colour, and are located in high altitude clouds [WCloud14]
- **Cirrocumulus clouds:** These clouds are high, thin. Cirrocumulus clouds are white. Looks similar to altocumulus clouds, but they are located at high altitude [WCloud14].
- **Cirrostratus clouds:** Cirrostratus clouds are very thin. They are similar to altostratus clouds, but are located at high altitude [WCloud14].

However, the 3D clouds are used to create stratus, cumulus, stratocumulus, nimbostratus, and cumulonimbus clouds. These types of clouds are located at low and middle altitude. Instead, cirrus, cirrostratus and cirrocumulus are located in high regions of the sky. For high-altitude cloud types, *CloudyDay* provides a 2D cloud approach. The 2D clouds are configured as follows:

```
std::vector<osgCloudyDay::Cloud2DState> cloud2dstates;

osgCloudyDay::CirrusStratusCloudState cirrus = osgCloudyDay::CirrusStratusCloudState();
//osgCloudyDay::CirrusCloudState cirrus = osgCloudyDay::CirrusCloudState();
//osgCloudyDay::CirrusCumulusCloudState cirrus = osgCloudyDay::CirrusCumulusCloudState();
//osgCloudyDay::AltStratusCloudState cirrus = osgCloudyDay::AltStratusCloudState();

cirrus.setMiddlePoint(osg::Vec3(0.f, 0.f, 5000.0f));
cirrus.setSize(osg::Vec2(50000.0f, 50000.0f));

cloud2dstates.push_back(cirrus);
```

Besides the representation of 2D clouds with a simple 2D texture, *CloudyDay* also provides a cloud generator of altocumulus and cirrus-cumulus clouds using a *Perlin* noise generator. To create a Perlin noise cloud, we use following code:

```
std::vector<osgCloudyDay::Cloud2DState> cloud2dstates;
osgCloudyDay::PerlinCloudState cirruscumolus = osgCloudyDay::PerlinCloudState();
cirruscumolus.setSharpness(0.9f);
cirruscumolus.setCover(0.25f);

cirruscumolus.setMiddlePoint(osg::Vec3(0.f, 0.f, 5000.0f));
cirruscumolus.setSize(osg::Vec2(50000.0f, 50000.0f));

cloud2dstates.push_back(cirrus);
```

The clouds, created with an Perlin noise generator, can be used similar to the simple texture approach. We only have to define the sharpness and the cover attribute. The *sharpness* attribute defines the size of the resulting clouds. The *cover* attribute defines the number of clouds.

Configuration of rain effect

Figure 9: Rain effect with fog.

Additionally, *CloudyDay* also supports rain. To configure this phenomenon, we use following code:

```
osgCloudyDay::RainState* rain = new osgCloudyDay::RainState();
rain->SetNumberOfParticles(100000);
rain->SetVelocity(osg::Vec3(0.f, 0.f, -1.f));
rain->SetPosition(osg::Vec3(0.f, 0.f, 1000.f));
rain->SetSize(osg::Vec3(2000.f, 2000.f, 1000.f));
Scene::m_rain = rain;
```

The number of particles attribute defines the density of the rain. The velocity defines the velocity of the rain. The rainy region can be defined with the attributes *position* (the middle point) and the *size*(size of the region).

Furthermore, the second parameter of the *CloudyDay* constructor has set to true. An example is given in the next line:

```
my_scene = new CloudyDay(terrain_config, true, false, false, true, true);
```

Configuration of fog

Also a fog effect is supported. We use following code to configure the effect:

```
osgCloudyDay::Fog* fog = new osgCloudyDay::Fog();
fog->SetFogColour(osg::Vec3(0.5f, 0.5f, 0.5f));
fog->SetFogDensity(0.1f);
```

We need to define the fog colour and the fog density. The *colour* attribute defines the colour of the fog. The *density* parameter defines the thickness of the fog. However, we can also use the 3D stratus cloud layer to create fog (as see above).

The last step is to pass the fog to the other objects. Each object (terrain, atmosphere, cloud, 3D model,...) has a *SetFog()* method. We only need to pass the fog object, as shown in the next line:

```
Scene::m_skydome->SetFog(fog);
```

Configuration of 3D models

Another important part is to add 3D models to the scene. A model can be added using following code:

```
std::vector<std::string> reflection_models;
reflection_models.push_back("DR400_Robin/bx2.obj");

std::vector<std::string> reflection_tex;
reflection_tex.push_back("DR400_Robin/bx2.bmp");

std::vector<int> ids_model;
ids_model.push_back(1);
```

To add an object, we add the paths to the models in a container. Then, we add the ids, needed for colour grading (this effect is presented in the next section).

Configuration of Post-Processing effects

Figure 10: Glare effect

For setting the post-processing effects the last two parameters of the *CloudyDay* constructor has to be set to true. The last but one parameter enables the glare effect, and the last parameter enables the star effect. An example is given in the next line:

```
my_scene = new CloudyDay(terrain_config, false, false, false, true, true);
```

Furthermore, we can add LUTs for colour grading. To add an LUT we use following code:

```
std::vector<std::string> luds;
luds.push_back("../data/lud/neutralLUT.bmp");
luds.push_back("../data/lud/yellowLUT.bmp");
```

As presented in the previous section, for each object an identifier can be assigned to determine the used LUT. The container has to pass at the Initialize method:

```
my_scene->Initialize(800,600,clouds, cloud2dstates, ids_model, reflection_models,
reflection_tex, fog, luds);
```

Setup scene

The last step is to setup the scene. We use following code:

```
my_scene = new CloudyDay(terrain_config, false, false, false, true, true);
my_scene->Initialize(800,600,clouds, cloud2dstates, ids_model, reflection_models,
reflection_tex, fog, luds);

osg::ref_ptr<osg::Group> root (new osg::Group);

root->addChild(my_scene->GetLightCamera().get());
root->addChild(my_scene->GetViewCamera2());
```

```
root->addChild(my_scene->GetLightCloudCamera().get());
root->addChild(my_scene->GetBlurShadowMapCamera().get());
//root->addChild(my_scene->GetLightshaftCamera());

root->addChild(my_scene->GetViewCamera());
root->addChild(m_lightCloudCamera.get());

root->addChild(my_scene->GetHUD().get());
if(my_scene->UseBloom() || my_scene->UseStar())
       root->addChild(my_scene->GetBlurProcess().get());
if(my_scene->UseBloom() || my_scene->UseStar())
       root->addChild(my_scene->GetBlur2Process().get());
root->addChild(my_scene->GetLuminanceCalculation().get());
root->addChild(my_scene->GetPostProcess().get());

osgViewer::Viewer viewer;
viewer.setCamera(my_scene->GetViewDepthCamera());
```

The first step is to create the *CloudyDay* object. As presented earlier, we need to pass the *TerrainConfig* object . Moreover, we need to define some attributes, which determine the post-processing effects. Then, we create a root object, and add the cameras to the root object. We add a camera for the sun, for the viewer, and for the clouds. If the shadow map should be blurred, we have to add the *BlurShadowMap* camera additionally. The last step is to add the cameras for the post-processing effects.

Modifying the scene at run-time

At runtime, some parameters can be modified. The sun position can be set as follows:

```
switch(ea.getEventType())
{
       case 'j' :    helperkey=1; sincosPos.x() += b;
                     break;
       case 'l' :    helperkey=2; sincosPos.x() -= b;
                     break;
       case 'k' :    helperkey=3; sincosPos.y() -= b;
                     break;
       case 'i':     helperkey=4; sincosPos.y() += b;
                     break;
}

osg::Vec4 value = osg::Vec4(0.f, 0.f, 0.f, 1.f) *
osg::Matrix::inverse(Scene::GetLightCamera()->getViewMatrix());

osg::Vec3f eye = osg::Vec3f();
osg::Vec3f center = osg::Vec3f();
osg::Vec3f up = osg::Vec3f();
Scene::GetLightCamera()->getViewMatrixAsLookAt(eye, center, up);

center = osg::Vec3(0.f, 0.f, 0.f);

osg::Vec3f center2eye = eye-center;
float length = center2eye.normalize();;
center2eye.x() = sinf(sincosPos.y())*cosf(sincosPos.x());
center2eye.y() = sinf(sincosPos.y())*sinf(sincosPos.x());
center2eye.z() = cosf(sincosPos.y());

if(center2eye.x() == 0.f && center2eye.y() == 0.f && center2eye.z() == 1.f)
{
       switch(helperkey)
       {
```

```
        case 1 :        sincosPos.x() += b;
                        break;
        case 2 :        sincosPos.x() -= b;
                        break;
        case 3 :        sincosPos.y() -= b;
                        break;
        case 4 :        sincosPos.y() += b;
                        break;
        }
        center2eye.x() = sinf(sincosPos.y())*cosf(sincosPos.x());
        center2eye.y() = sinf(sincosPos.y())*sinf(sincosPos.x());
        center2eye.z() = cosf(sincosPos.y());
}

center2eye.normalize();
center2eye *= length;

osg::Vec3 binm = osg::Vec3(0.f, 1.f, 0.f);
osg::Vec3 dir = center2eye;
binm.normalize();
osg::Vec3 h = binm^dir;
h.normalize();

osg::Vec3f lightPos = center2eye+center;
Scene::GetLightCamera()->setViewMatrixAsLookAt(lightPos, center, h);
Scene::GetCloudCamera()->setViewMatrixAsLookAt(lightPos, center, h);
Scene::m_skydome->SetLightPosition(lightPos);

osgCloudyDay::SkydomeMie* skymie =
dynamic_cast<osgCloudyDay::SkydomeMie*>(Scene::m_skydome);
if(skymie != 0)
{
        osgCloudyDay::CloudScene::sunLightColour = skymie
              ->CalculateSunColour(Scene::GetLightCamera()->getViewMatrix());

        my_scene->UpdateLightForClouds();
}
```

To enable/disable light scattering at clouds, we use following code:

```
osgCloudyDay::CloudGeometry::blur = true;
```

Integration of parts into an existing project

Figure 11: Sunset

In the previous chapter, we present a guide to configure *CloudyDay* to create a new scene. However, this plug-in can also be used to integrate only parts into an existing project. In the following, we present each step in detail to integrate some parts of *CloudyDay* into an existing project.

Creating the atmosphere model (required for terrain, and clouds)

The first step is to create a atmosphere model. This step is also required, if a terrain or clouds should be created. The first step is to configure the atmosphere model, as presented earlier:

```
Scene::m_skydome = new osgCloudyDay::SkydomeHimmel();
```

Then, we need to create the atmosphere. For the O'Neal [O'N05] approach, we use following code:

```
osgCloudyDay::SkydomeMie* skymie =
dynamic_cast<osgCloudyDay::SkydomeMie*>(Scene::m_skydome);
m_atmopshere = new osgCloudyDay::AtmosphereMie();
//m_atmopshere->SetFog(m_fog);
m_atmopshere->Initialize();
root->addChild(m_atmopshere->GetNode());
```

We only need to create a *AtmosphereMie* object and we have to initialize it. Then, we need to add the resulting geode to our camera. If we want to use the approach from Bruneton et al. [Bruneton 08], we use following code:

```
m_precompute = new osgCloudyDay::AtmospherePrecompute();
m_precompute->compute();

osgCloudyDay::AtmosphereHimmel* atmohimmel = new osgCloudyDay::AtmosphereHimmel();
atmohimmel->m_inscatter = m_precompute->getInscatterTexture();
atmohimmel->m_irradiance = m_precompute->getIrradianceTexture();
atmohimmel->m_transmittance = m_precompute->getTransmittanceTexture();

osgCloudyDay::SkydomeHimmel::m_inscatter = m_precompute->getInscatterTexture();
osgCloudyDay::SkydomeHimmel::m_irradiance = m_precompute->getIrradianceTexture();
osgCloudyDay::SkydomeHimmel::m_transmittance =m_precompute->getTransmittanceTexture()

//m_atmopshere->SetFog(m_fog);
```

```
atmohimmel ->Initialize();
root->addChild(atmohimmel->GetNode());
```

First, we need to create a *AtmospherePrecompute* object. Then, we pre-compute the atmosphere with calling the *compute* function. The next step is to create a *AtmosphereHimmel* object and to set the pre-computed textures. Then, we create the *Atmosphere* object and call the *Initialize()* method. The resulting *Geode* is added to the camera.

Adding a terrain to the scene (Optional)

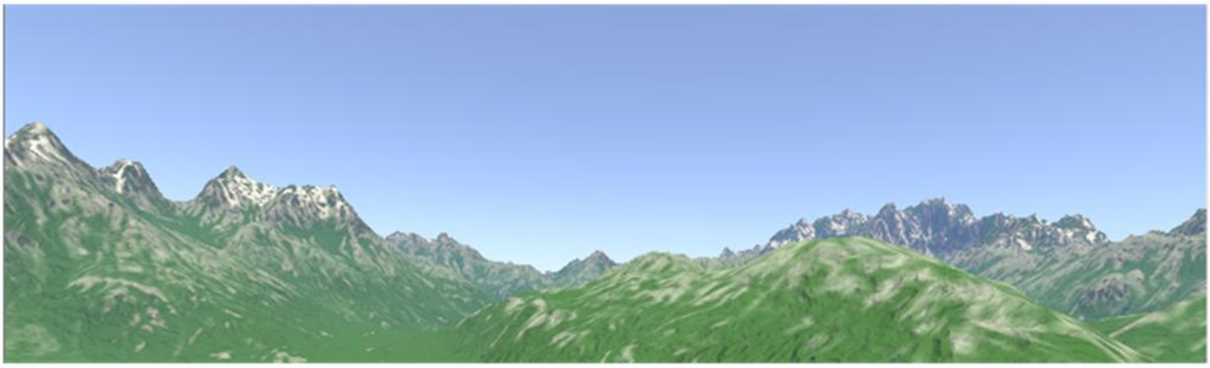


Figure 12: Atmosphere and terrain (Bruneton approach).

A terrain object can be added easily. The first step is to configure the terrain as presented earlier:

```
osgCloudyDay::TerrainConfig* terrain_config = new osgCloudyDay::TerrainConfig();
terrain_config->SetTesselationShader(false);

terrain_config->SetPath2File("mountains.obj");
terrain_config->SetPath2DiffuseTexture("mountain_diffuse.bmp");
terrain_config->SetPath2DefinationTexture("definition1.tga");
terrain_config->SetPath2HeightTexture("mountain_heightmap.tga");

terrain_config->AddPath2Texture("snow.bmp");
terrain_config->AddPath2Texture("terrain_rock4.bmp");
terrain_config->AddPath2Texture("terrain_grass.bmp");
terrain_config->AddPath2Texture("boulder.bmp");
terrain_config->AddPath2NormalTexture("rock_bump6.tga");
terrain_config->AddPath2NormalTexture("rock_bump4.tga");
```

To add a atmosphere, shaded with O'Neals atmosphere model [O'N05], we use following code.

```
if(m_terrainconfig->UseTesselationShader())
	m_terrain = new osgCloudyDay::TerrainGeometry(m_terrainconfig);
else
	m_terrain = new osgCloudyDay::TerrainMIE(m_terrainconfig);

m_terrain->SetFog(m_fog);
m_terrain->Initialize();

scene->addChild(m_terrain->GetNode());
```

The first step is to create a *TerrainMIE* or *TerrainGeometry* object, depending the value of *tesselation* attribute. Then, we need to call the *Initialize()* method. Then, we add the resulting *Geode* to the camera. To create a terrain with the Bruneton approach [Bruneton 08], we use following code:

```
osgCloudyDay::TerrainHimmel* m_terrainhimmel =  new
osgCloudyDay::TerrainHimmel(m_terrainconfig, m_terrainconfig->UseTesselationShader());
```

```
m_terrain = m_terrainhimmel;
m_terrain->SetFog(m_fog);
m_terrain->Initialize();

scene->addChild(m_terrain->GetNode());
```

The code is similar to the O'Neal approach [O'N05]. We only use a *TerrainHimmel* object.

Adding 3D clouds to an existing scene (Optional)

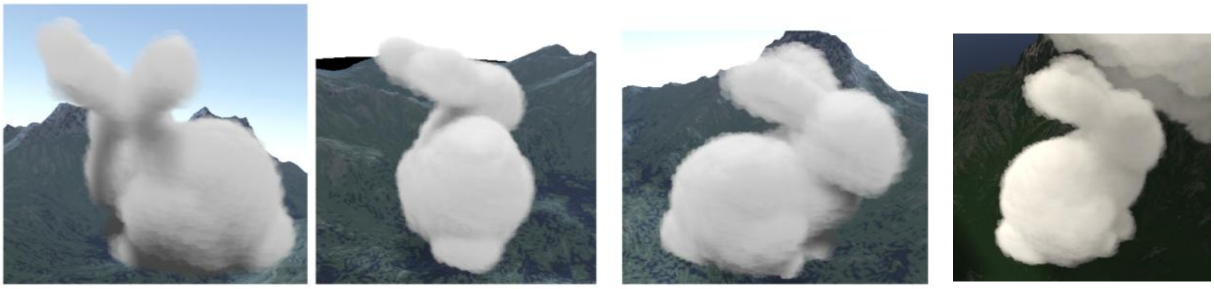


Figure 11: Stanford bunny rendered with our 3D cloud algorithm.

The first step of creating 3D clouds is to configure it. To configure a 3D cloud layer, we use following code:

```
osgCloudyDay::CloudScene::GetStates()->AddLayer(0, osgCloudyDay::CloudScene::CT_Cumulus);
osgCloudyDay::CloudScene::GetStates()->setOvercast(0, 0.002551f/1.f);
osgCloudyDay::CloudScene::GetStates()->setMiddlePoint(0, osg::Vec3(0.f,0.f, 2000.f));
osgCloudyDay::CloudScene::GetStates()->setSize(0, osg::Vec3(10000.f, 10000.f, 10.f));
osgCloudyDay::CloudScene::GetStates()->setClot(0, 10);
osgCloudyDay::CloudScene::GetStates()->setVariance(0, 0.125f);
osgCloudyDay::CloudScene::GetStates()->setColour(0, osg::Vec4(1.f, 1.f, 1.f, 1.f));
```

This code is equal as presented earlier in this report. To add a particular cloud, we use following code:

```
osgCloudyDay::CloudState* clouds = new osgCloudyDay::CloudState();
clouds->AddCloud(osgCloudyDay::CloudScene::CT_Cumulus, "Model/standford_bunny.obj",
       osg::Vec3(0.f,0.f,2125.f), osg::Vec3(100.f, 100.f, 100.f),
       CloudState::CStG_Voxel, osg::Vec4(1.f, 1.f, 1.f, 1.f));
```

As seen in this line, the configuration of the cloud is equal. To create the 3D clouds we have to add following code:

```
my_scene = new Scene();
m_scene->CreateCloudCamera();
m_cloudcreator = new osgCloudyDay::CloudCreator(scene);
m_cloudcreator->Initialize(clouds);

root->addChild(my_scene->GetLightCloudCamera().get());
```

The first step is to create a scene object and a camera to render the 3D clouds. The next step is to create a *CloudCreator* object, and we need to initialize it. The last step is to add the camera to the scene. The 3D cloud approach renders into two textures:

```
CloudScene::fbo_cloud_texture //result from viewer
CloudScene::fbo_light_texture //result from light source (shadow map)
```

These textures can be used to integrate the3D clouds into our own OSG project.

Adding 2D clouds to a existing scene (Optional)

The 2D clouds can also be added to an existing scene. We use following code:

```
osgCloudyDay::AltStratusCloudState cirrus = osgCloudyDay::AltStratusCloudState();
cirrus.setMiddlePoint(osg::Vec3(0.f, 0.f, 5000.0f));
cirrus.setSize(osg::Vec2(50000.0f, 50000.0f));

osgCloudyDay::Create2DCloud* m_create2dclouds = new osgCloudyDay::Create2DCloud();
m_create2dclouds->Initialize(cirrus);

root->addChild(m_create2dclouds->GetNode());
```

First, we need to configure the 2D clouds. This code is equal to the code presented in the previous chapter. Then, we need to create a *Create2DCloud* object. We need to pass the configuration of the 2D clouds and to call the *Initialize()* method. The last step is to add the resulting *Geode* to the camera.

Adding rain to a scene (Optional)

Figure 13: Rain and fog

Rain can be added very easily to an existing project. We use following code to add rain:

```
osgCloudyDay::RainState* rain = new osgCloudyDay::RainState();
rain->SetNumberOfParticles(100000);
rain->SetVelocity(osg::Vec3(0.f, 0.f, -1.f));
rain->SetPosition(osg::Vec3(0.f, 0.f, 1000.f));
rain->SetSize(osg::Vec3(2000.f, 2000.f, 1000.f));
Scene::m_rain = rain;

osgCloudyDay::Rain::CreateShader();
osgCloudyDay::Rain::CreateTexture();
osgCloudyDay::Rain* m_rain = new osgCloudyDay::Rain();
m_rain->Initialize();

root->addChild(m_rain->m_geode);
```

First, we need to create the shader and the 3D raindrop texture. Then, we create a *Rain* object and we need to call the *Initialize()* method. The last step is to add the *Geode* to the existing scene.

Adding post-processing effects to a scene (Optional)

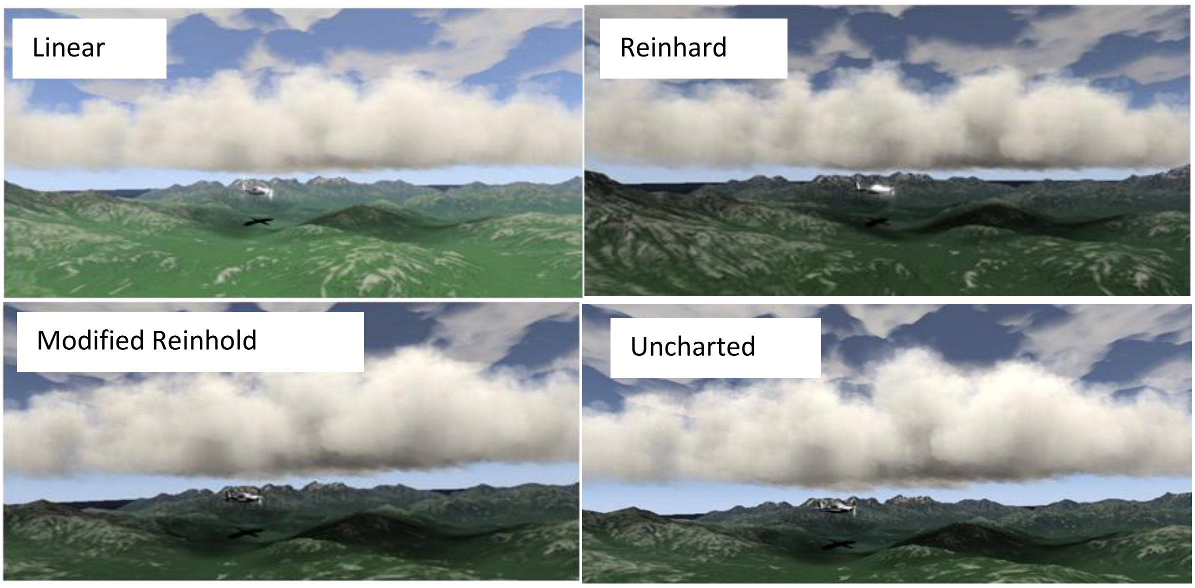


Figure 13: Results of different HDR mapping algorithms.

The post-processing effects can also be added to an existing scene. We use following code:

```
m_hud = new osgCloudyDay::HUD();
m_hud->SetSceneTexture(osgCloudyDay::Scene::GetSceneTexture());
m_hud->SetGodrayTexture(osgCloudyDay::Scene::GetGoodRayTexture());
m_hud->SetCloudTexture(osgCloudyDay::CloudScene::fbo_cloud_texture);

osgCloudyDay::HUD::Initialize();
m_hud->CreateCamera();

m_blur = new osgCloudyDay::Blur(4);
m_blur->CreateCamera();
m_blur2 = new osgCloudyDay::Blur(5);
m_blur2->CreateCamera();

m_luminance = new osgCloudyDay::LuminanceCalculation();
m_luminance->CreateCamera();

m_postprocess = new osgCloudyDay::PostProcess(hdr_mapping, use_avglum, m_bloom, m_star);
for(unsigned int i = 0; i < luds.size(); i++)
       m_postprocess->AddLUT(luds[i]);
m_postprocess->CreateCamera();

m_hud->CreateGeometry();
m_blur->CreateGeometry();
m_blur2->CreateGeometry();
m_postprocess->CreateGeometry();
m_luminance->CreateGeometry();

root->addChild(my_scene->GetHUD().get());
if(my_scene->UseBloom() || my_scene->UseStar())
       root->addChild(my_scene->GetBlurProcess().get());
if(my_scene->UseBloom() || my_scene->UseStar())
       root->addChild(my_scene->GetBlur2Process().get());
root->addChild(my_scene->GetLuminanceCalculation().get());
root->addChild(my_scene->GetPostProcess().get());
```

First, we need to create a *HUD* object. Before initialization, we have to set the scene, good-ray and cloud textures. Then, we add 2 *Blur* (vertical and horizontal blur), a *LuminanceCalculation,* and a *PostProcessing* object. At creation of the *PostProcessing* object, we pass the HDR Mapping method, and we define, if the average luminance, the bloom, and star effect should be calculated. We also need to create a camera and a screenquad for each object. The last step is to add the cameras to the scene.

References:

[Bouthors08] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. 2008. Interactive multiple anisotropic scattering in clouds. In Proceedings of the 2008 symposium on Interactive 3D graphics and games (I3D '08). ACM, New York, NY, USA, 173-182.

[Bruneton 08] Bruneton, E. and Neyret, F. (2008), Precomputed Atmospheric Scattering. Computer Graphics Forum, 27: 1079–1086.

[Crane07] Crane, Keenan and Llamas, Ignacio, and Tariq, Sarah: "Real Time Simulation and Rendering of 3D Fluids", 2007, GPUGems 3, chapter 30, Addison-Wesley

[Dobashi00] Y.Dobashi, K.Kaneda, H.Yamashita, T.Okita, T.Nishita, "A Simple, Efficient Method for Realistic Animation of Clouds," Proc. SIGGRAPH2000, 2000-7, pp. 19-28.

[Donnelly06] Donnelly, William, and Andrew Lauritzen. 2006. "Variance Shadow Maps." In Proceedings of the Symposium on Interactive 3D Graphics and Games 2006, pp. 161–165.

[Kshitiz06] Kshitiz Garg and Shree K. Nayar. 2006. Photorealistic rendering of rain streaks. ACM Trans. Graph. 25, 3 (July 2006), 996-1002

[Green08] Simon Green: "Volumetric Particle Shadows", NVIDIA Whitepaper , 2008

[Harris01] Mark J. Harris and Anselmo Lastra, Real-Time Cloud Rendering. Computer Graphics Forum (Eurographics 2001 Proceedings), 20(3):76-84, September 2001

[Kaneko01] Tomomichi Kaneko and Toshiyuki Takahei and Masahiko Inami and Naoki Kawakami and Yasuyuki Yanagida and Taro Maeda and Susumu Tachi: " Detailed shape representation with parallax mapping", In Proceedings of the ICAT 2001, 2001

[Luksh07] Luksh, C.: Realtime HDR rendering. Tech. Rep., Institute of Computer Graphics and Algorithms, TU Vienna (2007)

[Mit07a] MITCHELL K.: Volumetric Light Scattering as a Post-Process. Addison-Wesley, 2007, pp. 275–285.14

[Müller12] Daniel Müller, Juri Engel, Jürgen Döllner: Single-Pass Rendering of Day and Night Sky Phenomena. VMV 2012: 55-62

[O'N05] O'NEILS.: Accurate atmospheric scattering. In GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation(2005), Addison-WesleyProfessional.

[Nishita99] Nishita, T.; Dobashi, Y., "Modeling and rendering methods of clouds," Computer Graphics and Applications, 1999. Proceedings. Seventh Pacific Conference on , vol., no., pp.218,219, 326, 1999

[Perlin02] K.Perlin. Improving noise. In Proceedings of the 29th annual conference on Computer graphics and interactive techniques. ACM Press, 2002.

[Tariq07] S Tariq: "Rain", NVIDIA Whitepaper, 2007

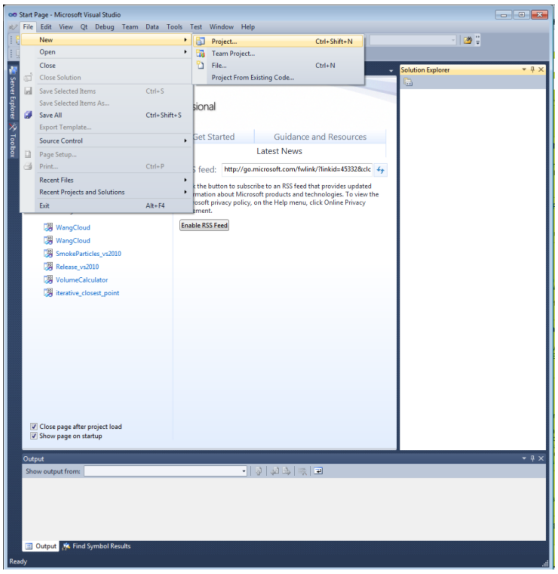
[Wang03] Niniane Wang. 2003. Realistic and fast cloud rendering in computer games. In ACM SIGGRAPH 2003 Sketches & Applications (SIGGRAPH '03). ACM, New York, NY, USA, 1-1.

[WCloud14] Wikipedia: "Cloud", 2014, Available at: http://en.wikipedia.org/wiki/Cloud (date: 1.10.2014)

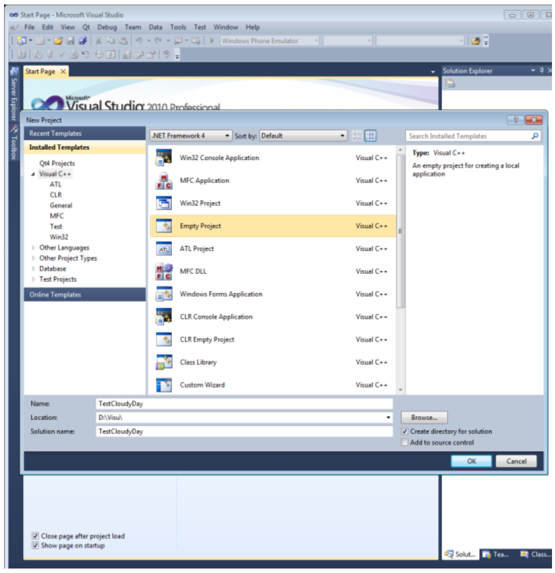
[Zendel13] O. Zendel, W. Herzner, and M. Murschitz. Vitro - model based vision testing for robustness. Proceedings to the 44th International Symposium on Robotics - ISR 2013, pages October 24–26, 2013

Appendix A - Creation of a new Project

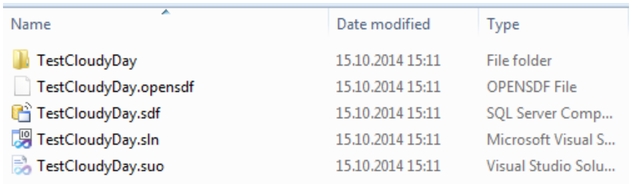
In this chapter, we show all steps to create a new project in detail.



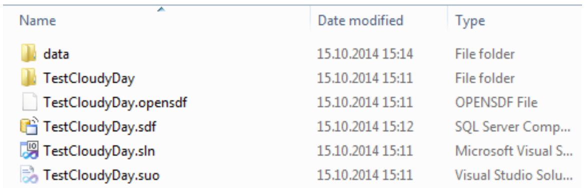
Create a new project



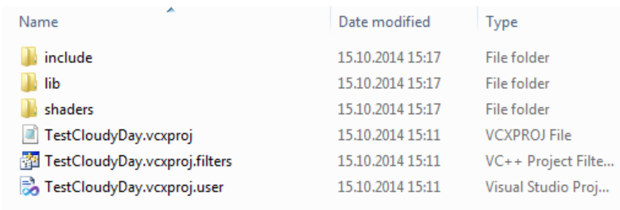
Select "Empty Project"



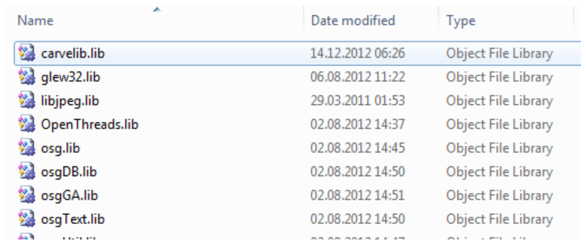
Go to the created directory



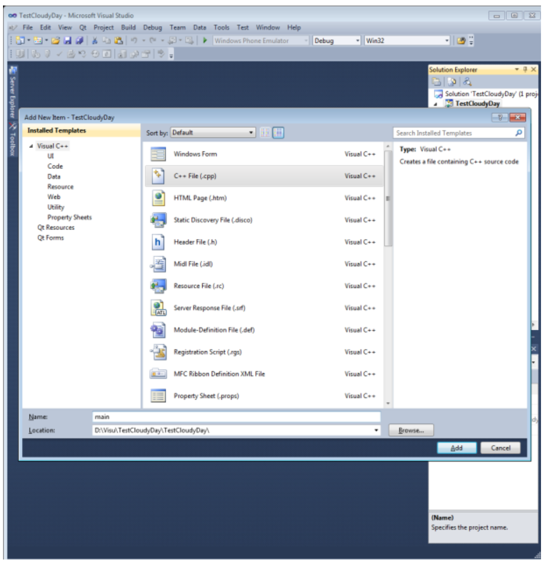
Add the data directory from the archive



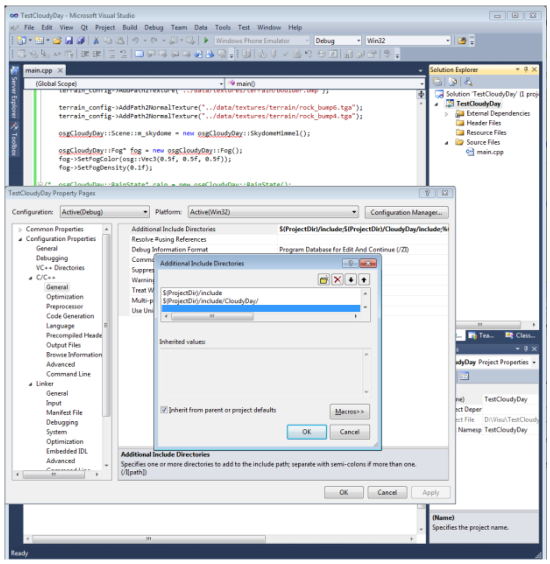
*Add include, library and shader directory to the directory, that contains the *. vcxproj file.*



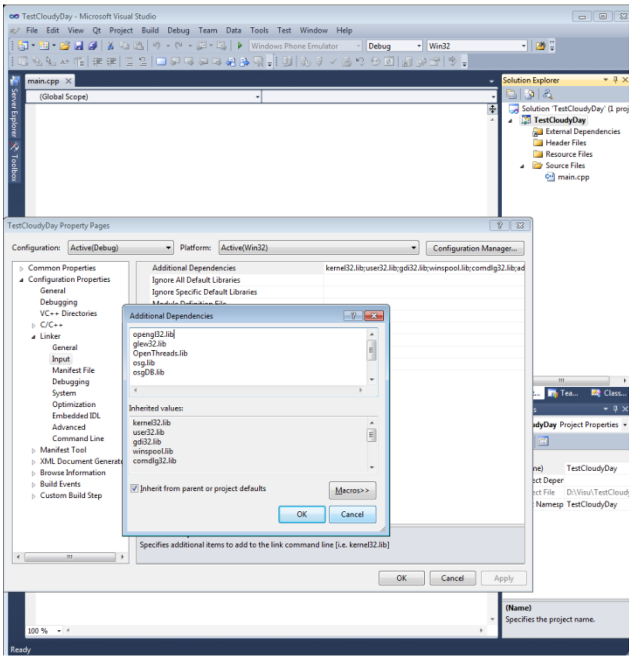
Add OSG libraries and include files to the project



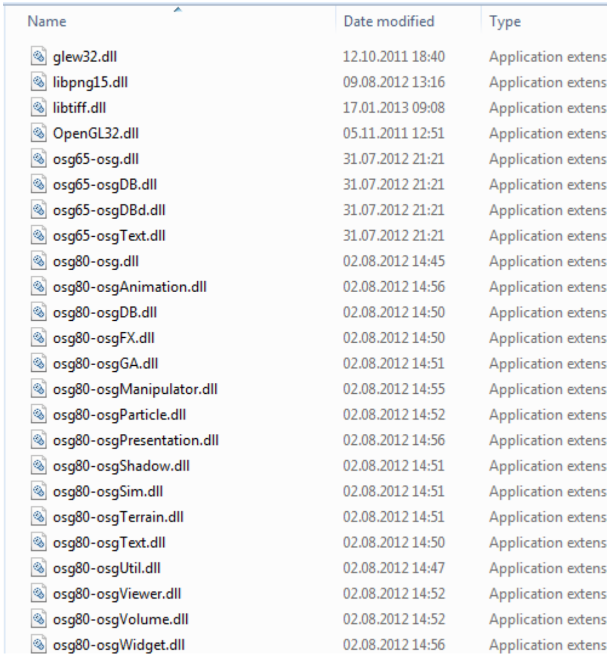
*Add a *.cpp file*



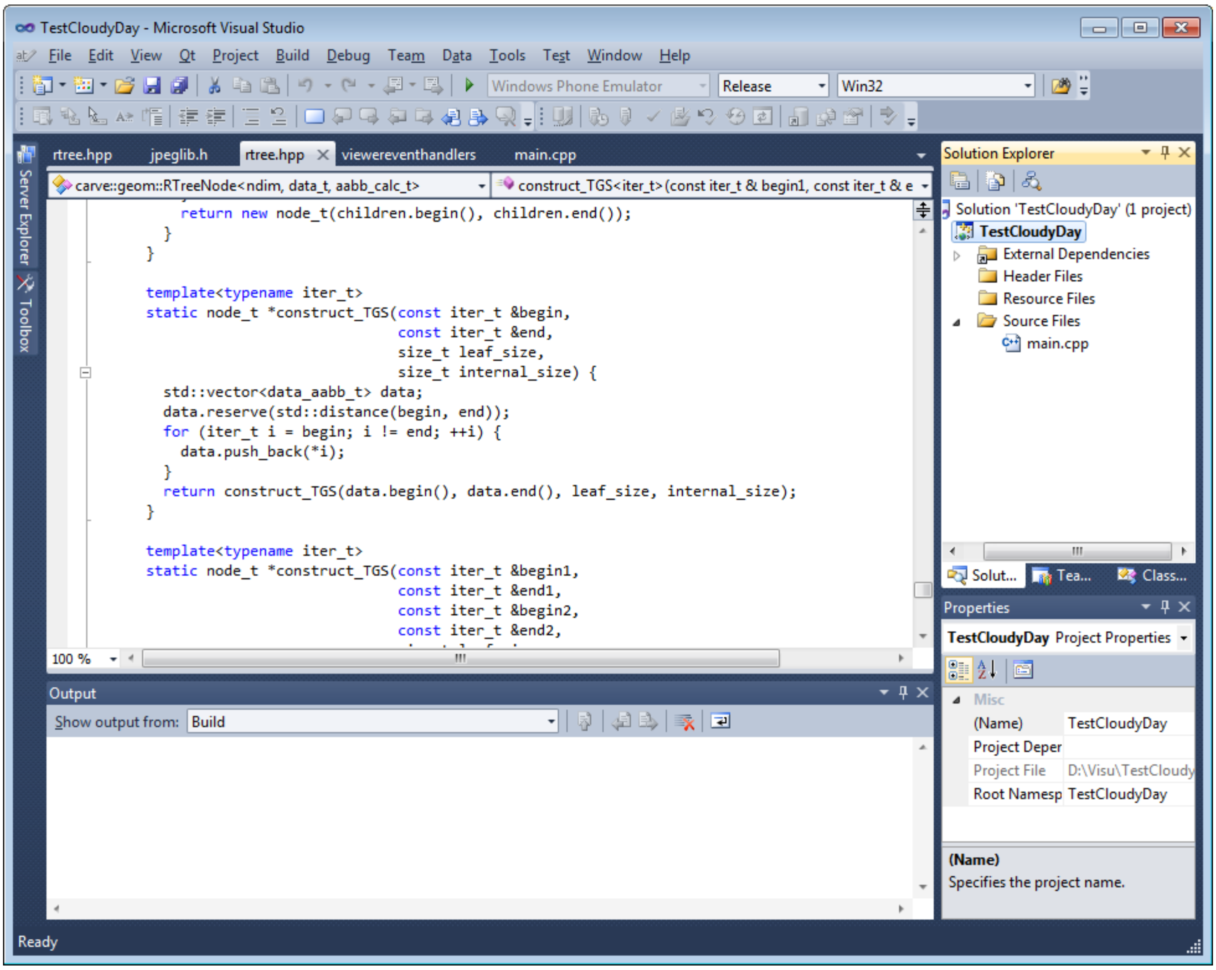
Set the Include and library paths



Define the additional dependencies



*Add *.dll files in the release directory.*



Ready to code!