Particle Universe Manual

- Plugin -

Author: Henry van Merode
Version 1.5.1
www.fxpression.com

Introduction

Particle Universe is a complete system to create visually stunning particle systems for Ogre[1]-powered video games and video editting. The package consist of an editor for creation of particle systems using a visual editting system, and a runtime plugin for in-game usage of the created particle system scripts.

This manual is used to setup the plugin and describes the AtlasImage tool; this tool is used to create an Atlas image.

[1] Ogre is a multiplatform rendering system that is widely used in commercial video games. See www.ogre3d.org

The Particle System plugin

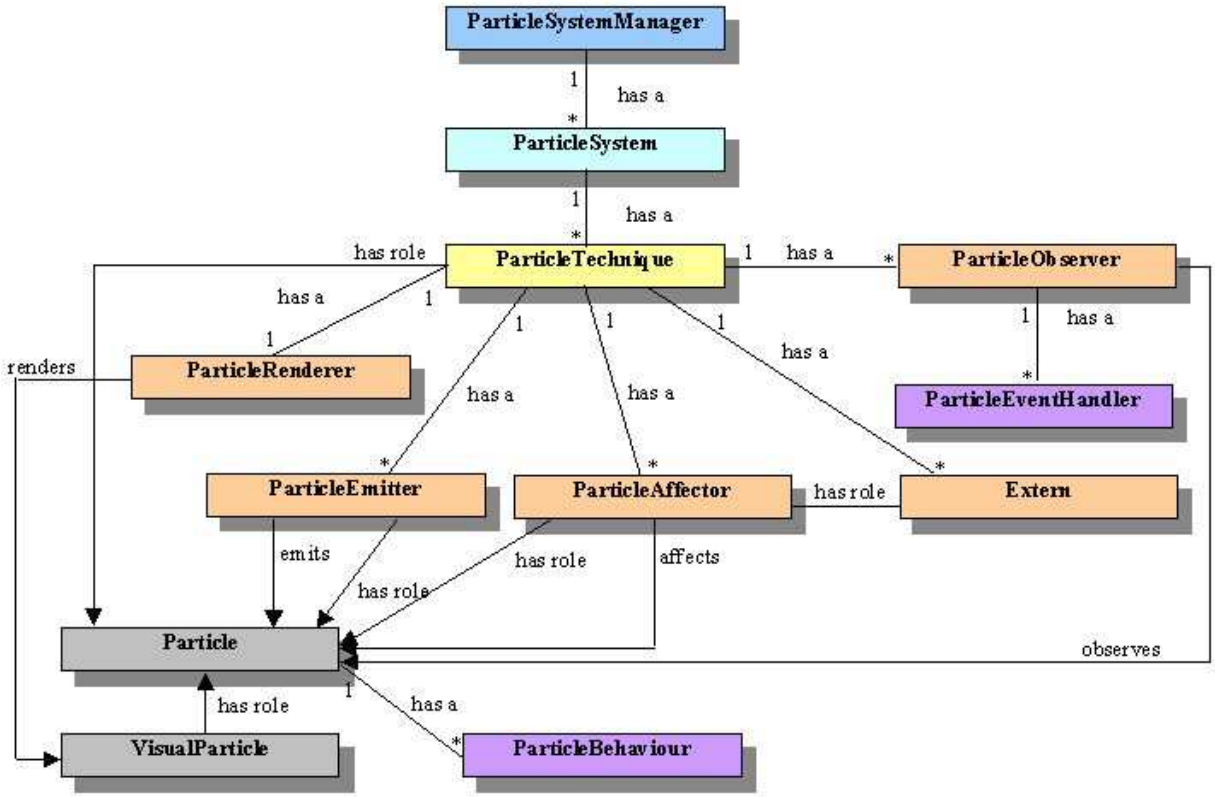
The Particle Universe plugin is a DLL (in case of Windows™) that can be loaded in the Ogre render engine. The Visual Studio solution files (.sln) and all C++ code is included in the package. The directory structure of the plugin reflects the directory structure of Ogre.

> ⓘ
>
> *In older versions of the Particle Universe plugin, one of the subdirectories contained a media directory, including all material, script and mesh files. These files are now included as part of the editor. The only media that is left for the plugin concerns the core shaders.*

The API reference – in HTML format – is included in one of the plugin subfolders (\Docs)

Class Diagram

The figure below presents the main classes of which the Particle Universe plugin consist. Most of the components are described in the Editor manual.

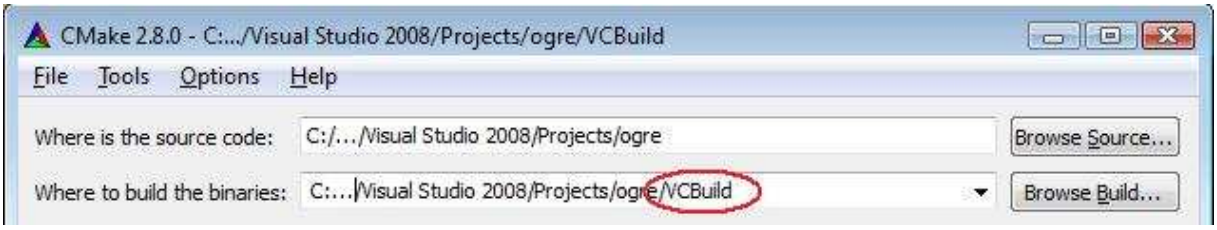


Setup plugin

Install the Particle Universe package by starting the Window installer. This results in the creation of the following directory structure:

The *Particle Universe Editor* directory contains the executable of the editor. The *Particle Universe Plugin* directory contains a *VCBuild* directory. With the introduction of Ogre version 1.7 the way the Ogre package is build, has been changed. Building Ogre is performed by means of CMake and the whole generation/building of the Ogre SDK is done in a separate folder.

When building the Ogre components, assume that you selected one of the Visual Studio compilers in the CMake dialog and generated the necessary files to the *VCBuild* directory as shown in the CMake dialog below.



Of course this may be a complete different directory, but for convenience it is assumed that the directory is named *VCBuild* (and so is the directory in the Particle Universe package).

Copy all subdirectories and files of *VCBuild* from the *Particle Universe Plugin* directory to the *VCBuild* directory of Ogre, that was generated with CMake (which contains all the Ogre code). All Particle Universe files are copied to the Ogre directory strucure.

Next step is compiling the Particle Universe DLL. Open ***ParticleUniverse_vc9.sln*** in *VCBuild* and compile. The 'post-event build' copies some files to Ogre's *sdk* directory. The *sdk* directory is used by applications that make use of both Ogre and Particle Universe.
After compilation, the *ParticleUniverse_d.dll* and *ParticleUniverse.dll* can be used similar to Ogre's own plugins. For usage, add the entry ***Plugin=ParticleUniverse_d*** to the *Plugins.cfg* file of your application if you work in a Debug mode, or add ***Plugin=ParticleUniverse*** for a Release build.

> ⓘ
> *When you want to use the 'soft particles' option, the required shader files must be used. This means that the directory* ***VCBuild\sdk\media\ParticleUniverse\core*** *must be added to your* ***resources.cfg*** *file.*
>
> *Using 'soft particles' takes some fiddling. Also experiment with depthscale in your code using the setDepthScale() function:*

```
ParticleUniverse::ParticleSystemManager::getSingletonPtr()->setDepthScale(10.0);
```

PhysX™

Setup PhysX™

If the PhysX engine is used in Particle Universe, you have to:

1. Uncomment the line *#ifdef PU_PHYSICS* in ***ParticleUniversePrerequisites.h*** (by default the PhysX engine is supported in Particle Universe, but other physics engines can also be added, although this still takes a decent amount of code).
2. Set the environment variable ***PHYSX_HOME***. For PhysX version 2.8.1. this can be done by the command:
   set PHYSX_HOME=C:\Program Files\NVIDIA Corporation\NVIDIA PhysX SDK\v2.8.1\SDKs
   The ***PHYSX_HOME*** environment variable is used in the compiler settings.
3. Add ***PhysXLoader.lib*** to the linker settings. Linking to ***PhysXLoader.lib*** can be omitted if the client application itself already links to this lib.
4. The project properties contains references to PhysX™ directories (*$(PHYSX_HOME)Physics\include*, for example). If you don't want to use Physx™ you can remove them or just keep them in. If you keep the directories, the compilation returns a *Project : warning PRJ0018*. This is harmless, so there is no urgent reason to remove these references.

The compatibility matrix below lists which version of the Particle Universe plugin works with the version of Ogre and PhysX™.

Particle Universe version	Ogre version	PhysX™ version
1.5.1	1.8	2.8.1
1.5	1.7	2.8.1
1.4	1.7	2.8.1
1.3	1.7	2.8.1
1.2	1.7	2.8.1
1.1	1.6	2.8.1
1.0 / 1.01	1.6	2.8.1
0.81	1.6	–
0.8	1.4.6	–

Using PhysX™

Particle Universe is prepared for easy integration of PhysX™. For usage of the PhysX™ features, Particle Universe contains a *PhysXBridge*, a singleton class that communicates with PhysX™. Particle Universe assumes that initialisation and simulation of PhysX™ is done in the client application, but for testing purposes, the *initNx()* and *exitNx()* functions are supported by Particle Universe. The *PhysXBridge* contains a few important functions that must be called from outside the Particle Universe plugin.

setScene(NxScene* scene) Setting the scene must be done before PhysX™ can be used in Particle Universe. This is done by means of calling *ParticleUniverse::PhysXBridge::getSingletonPtr()->setScene(myNxScene);*

synchronize(Ogre::Real timeElapsed) After each PhysX™ simulation step, Particle Universe needs to be synchronized. The *timeElapsed* argument is the time between two simulation steps.

onContactNotify(NxContactPair& pair, NxU32 events, NxVec3 contactPoint) Alhough Particle Universe includes an *NxUserContactReport* class for testing purposes, it is assumed that the client application also wants to use a *NxUserContactReport*. Therefore the assumption is that the *NxUserContactReport* of the client application is leading. To register particle collision in Particle Universe and to be able to act, a callback function must be made like the example below. The use of the *onContactNotify()* is optional. Only in situations where you want to use an *OnCollision...DoSomething* construction, you have to do a callback to the *onContactNotify()* function of the *PhysXBridge*. Beware that the *onContactNotify()* callback is only used for "Rigid body based" particles:

```
void MyContactReport::onContactNotify(NxContactPair& pair, NxU32 events)
{
    NxContactStreamIterator i(pair.stream);
    while(i.goNextPair())
    {
        while(i.goNextPatch())
        {
            while(i.goNextPoint())
            {
                PhysXBridge::getSingletonPtr()->onContactNotify(pair, events,
                   i.getPoint());
            }
        }
    }
}
```

Each particle system that uses the PhysX capabilities must add a *PhysX Extern* to its *Technique*. With the use of the *PhysX Extern*, the shape and other PhysX™ properties are assigned to the particles.

Create, start, stop and delete a particle system

Creation of a particle system is done by means of the *ParticleSystemManager*. If you want to create a particle system in your code, you must do this:

```
ParticleUniverse::ParticleSystemManager* pManager =
ParticleUniverse::ParticleSystemManager::getSingletonPtr();
ParticleUniverse::ParticleSystem* pSys = pManager->createParticleSystem("mySys",
    "nameOfTemplateScript", mSceneManager);
mNode->attachObject(pSys);
```

The "nameOfTemplateScript" is the name of a particle script in a *.pu file, identified as:

```
system nameOfTemplateScript
{
   ...
}
```

Depending on the situation, you can just start and stop the particle system, but between creation and starting a particle system, there is another step – the prepare step -, which is automatically executed in the *start()* functions. Some particle systems however have some time-consuming precalculation steps. It is adviced to perform these steps at the start of the scene creation to prevent framerate drops. This is done by means of the prepare() function.

```
ParticleUniverse::ParticleSystemManager* pManager =
ParticleUniverse::ParticleSystemManager::getSingletonPtr();
ParticleUniverse::ParticleSystem* pSys = pManager->createParticleSystem("mySys",
    "nameOfTemplateScript", mSceneManager);
mNode->attachObject(pSys);
pSys->prepare();
```

Beware, that if a particle system is started, the resources (materials, textures) may not be loaded up front and are loaded as soon as the particle system is started. This also can cause framerate hickups. This is not a problem of Particle Universe itself, but a generic issue of Ogre. Take care of preloading the materials at the creation of a scene if possible.

When a particle system is created, it can be started. Standard, the *start()* function is called, without arguments. The *start()* function 'starts' a particle system, until it is either actively stopped by one of the *stop()* functions or – if configured this way – automatically stopped by the particle system itself (i.e. by means of the *DoStopSystemEventHandler*). Also pausing and resuming a particle system is possible. There are a few variations of the start, stop, pause and resume functions, with gives more control over the particle systems life-cycle. The table below includes the variations:

Particle System function	Description
`start(void)`	Start a particle system (standard usage).

`start(Ogre::Real stopTime)`	Start a particle system that automatically stops after ‘stopTime’ seconds.
`startAndStopFade(Ogre::Real stopTime)`	Start a particle system, which stops emitting particles after ‘stopTime’ seconds; this applies to all emitters in the particle system. This function differs from the previous function in that sense that the *startAndStopFade()* function does not stop immediately, but gradually, after the last particle has been expired.
`stop(void)`	Stops a running particle system (standard usage).
`stop(Ogre::Real stopTime)`	Stops a running particle system after ‘stopTime’ seconds. Remark: Although there is a *start(Ogre::Real stopTime)* function that also stops after ‘stopTime’ seconds from start, you often have a situation where the particle system was just started with the *start()* function.
`stopFade(void)`	Stops the particle system from emitting particles.
`stopFade(Ogre::Real stopTime)`	Stops the particle system from emitting particles after ‘stopTime’ seconds.
`pause(void)`	Pauses a running particle system.
`pause(Ogre::Real pauseTime)`	Pauses a running particle system during a period of time. After this time, the particle system automatically resumes.
`resume(void)`	Resumes a paused particle system.

Example of creating and starting a particle system:

```
ParticleUniverse::ParticleSystemManager* pManager =
ParticleUniverse::ParticleSystemManager::getSingletonPtr();
ParticleUniverse::ParticleSystem* pSys = pManager->createParticleSystem(“mySys”,
    “nameOfTemplateScript”, mSceneManager);
mNode->attachObject(pSys);
pSys->start(3.0f);
```

Particle systems that where created by means of the *createParticleSystem()* function of the *ParticleSystemManager*, must also be deleted by means of the *destroyParticleSystem()* function. A good practice is to destroy particle systems before the complete scene is destroyed. A fast way of deleting all particle systems is by means of *ParticleSystemManager::destroyAllParticleSystems()*.

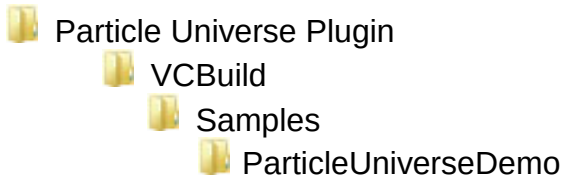
Particle systems must always be actively deleted, preferable before other scene objects are destroyed.

Example to delete a particle system:

```
pSys->stop();
mNode->detachObject(pSys);
pManager->destroyParticleSystem(“mySys”, mSceneManager);
```

Demo application

From version 1.5 a demo application is shipped with the package. This demo application runs in the Samplebrowser of Ogre 3D. The project is included in the ParticleUniverse solution and the code can be found in:



To run this demo, a few things must be done:

1.
Add the resources needed for this demo to *VCBuild/bin/release/resources.cfg* and to *VCBuild/bin/debug/resources.cfg*.

It makes use of the scripts and textures of the Particle Universe Editor. Add the following settings (the path may be different if the Particle Universe Editor is on another location):

```
FileSystem=../../ParticleUniverseEditor/media/examples/scripts
FileSystem=../../ParticleUniverseEditor/media/examples/materials
FileSystem=../../ParticleUniverseEditor/media/examples/models
FileSystem=../../ParticleUniverseEditor/media/examples/textures
```

2.
Add the entry

```
Plugin=ParticleUniverse_d
```

to *VCBuild/bin/debug/plugins.cfg*

Add the entry

```
Plugin=ParticleUniverse
```

to *VCBuild/bin/release/plugins.cfg*

3.
Add the entry
`SamplePlugin=Sample_ParticleUniverseDemo_d`
to *VCBuild/bin/debug/samples.cfg*

Add the entry
`SamplePlugin=Sample_ParticleUniverseDemo`
to *VCBuild/bin/release/samples.cfg*

4.

Compile the Sample_ParticleUniverseDemo (in debug and/or release). Make sure you compiled the ParticleUniverse plugin first.

5.
Run the SampleBrowser_d.exe (Debug mode) SampleBrowser.exe (Release mode).

Particle system events

An application that uses the Particle Universe plugin may act on a certain event that occurs with a particle system. Examples are

- playing a sound when a particle system starts.
- deleting a particle system when a particle system stops.

Sometimes the event is initiated by the application itself, for example starting a particle system. In other cases, the event occurs withing the particle system. It is possible to create and register one or more *ParticleSystemListeners* for each particle system, which listen to the particle system events. Each listener must implement the virtual function *handleParticleSystemEvent()*. In the implementation of this function, the possible events are handled. Below is a code example of a *Tank* class, which is a child of *ParticleSystemListener*, handling certain events:

```
void Tank::handleParticleSystemEvent(ParticleUniverse::ParticleSystem* particleSystem,
ParticleUniverseEvent& particleUniverseEvent)
{
   switch(particleUniverseEvent.eventType)
   {
      case PU_EVT_SYSTEM_STARTED:
      {
          // Tank fired a grenade
          playSound(“fireGrenade”, particleSystem->getDerivedPosition());
      }
      break;

      case PU_EVT_EMITTER_STOPPED:
      {
          // When the emission of particles stops, play a reload sound
          playSound(“reloadGrenade”,
                    particleUniverseEvent.emitter->getDerivedPosition());
      }
      break;
   }
}
```

Possible events are listed in the file ***ParticleUniverseCommon.h***

Note, that the particle system can also contain observers and event handlers. These type of objects also act on certain events, but these are usually events of individual particles. These types of components are also used inside a particle system, while the *ParticleSystemListener* is for external use.

Tools

Atlas Image Tool

The ***Atlas Image Tool*** can be found in the *Tools* directory and is used to create atlas images (textures), which are supported by the Particle Universe plugin. For example, an atlas texture can be used in combination with the *Texture Animator*. The *Texture Animator* (= particle affector) uses an atlas texture and uses rectangles of the texture (defined by texture coordinates) to create an animation.
The ***Atlas Image Tool*** is a commandline tool that reads a configuration file and creates an atlas image. Syntax:

```
AtlasImageTool    configFileName
```

Example of its usage:

```
C:\AtlasImageTool atlas.cfg
```

Creation of an atlas image

The configuration file contains the settings, needed to create an atlas texture. Example:

```
// Set the path of all image files
ImagePath = ../../images


// Define the input files
InputImage = flare.png; smoke.png; smoke.png


// Alpha correction
Alpha = 1.0; 1.0; 0.0


// Define the output (atlas) file
OutputImage = interpolate.png
```

The configuration file uses 3 images from 3 .png files and creates 1 new .png file. This new file contains the other 3 images. The ***Atlas Image Tool*** tries to distribute the input images as optimal as possible.

- ***ImagePath*** is a keyword and identifies the relative directory where the images are stored.
- ***InputImage*** is a keyword and identifies a list of imagefiles of which the atlas image is constructed. A restriction is that all imagefiles must have the same dimensions (width and height), the same format and no mipmaps.
- ***Alpha*** is an optional keyword and identifies a list of values. Each value is used to correct the alpha component of its related imagefile.

- ***OutputImage*** is a keyword and identifies the name of the atlas image that is created.

Interpolation between images

An optional feature is to generate image frames between the specified input images. For example, if image1.png and image2.png are defined with the ***InputImage*** keyword, and frame 0 and 10 are assigned to it (by means of the ***Frame*** keyword), image frames 1..9 are automatically generated. The images are interpolated between image1.png and image2.png.

```
// Define the input files
InputImage = image1.png; image2.png


// Relate every inputfile to a frame (intermediate frames are interpolated)
Frame = 0; 10
```

- ***Frame*** is a keyword and identifies a list of framenumbers that is related to the list with input images.

Runtime analysis

The file ***ParticleUniversePrerequisites.h*** contains the *#define PU_LOG_DEBUG* directive, that determines whether runtime debug info is on or off. It can be used to determine the optimal values of the quota's, defined in a *Technique*. If a *Particle System* runs and *PU_LOG_DEBUG* is defined, the maximum number of emitted visual particles, emitted emitters, emitted techniques and emitted *Affector*s, is calculated. When the particle system stops, this data is written to the .log file and can be used to set the quota's.

This will be integrated in a future release of the Particle Universe Editor.

Multi-threading

Multi-threading capabilities of Particle Universe are still limited, but the first steps are already taken. The *ForceField Affector*, that ships with version 1.3, allows generation of a force field in a separate thread. This is activated when the Forcefield type is set to ‘matrix’.

This option only works if OGRE_THREAD_SUPPORT (in OgreConfig.h) is set to a value > 0. Particle Universe makes use of the worker queue functionality in Ogre.

Future development

There is still a lot of room for improvement for the plugin. Some highlights[2]:

- Multi-threading.
- GPU rendering.
- Volumetric shadowing to particle systems.
- High-Speed, Off-Screen Particles.

[2] No guarantees when and if these features are realised.