Bachelor's Thesis

„Jungle Rumble“

Martin Knecht, 0326294, martin.knecht@aon.at

Michael Schwärzler, 0325222, michael@schwaerzler.com

SS 2005/2006

TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Index

Introduction

In our bachelor's thesis, we present a game called "Jungle Rumble" which was developed under supervision of the "Institute of Computer Graphics and Algorithms" at the Technical University of Vienna. The goal of the project was to create a real-time 3d-game which implements various effects developed and provided by the "GameTools Project" (GTP):

> "The **GameTools Project** is an EU project from the 6th Framework Programme that brings together **leading European computer graphic experts** from universities in Austria, France, Hungary and Spain with **European industrial partners** from the fields of computer game development and virtual reality to create **next generation realtime 3D libraries** for **Geometry**, **Visibility** and **Global Illumination** for the **PC platform**, with an extension to consoles **PS2, XBox, PS3, XBox 360** planned." (Quote from the GTP Website)

Since the TU Vienna is a GTP member, we were allowed to participate actively in the EU project by creating this game, supervised by Dipl. Ing. Markus Giegl and Dr. Michael Wimmer.

The main reason for creating this game was to use some of the developed effects in a "real" game, not only in small examples and demos. This is generally a quite challenging task, since there are lots of things which have to be processed in real time beside the special effects (for example game physics). We were able to include three of the technologies created by the GTP: An approximate raytracer, depth impostors and a special kind of depth impostors using a noise texture to create heat haze.

Furthermore we decided to use a professional physic engine in our game. We chose the PhysX engine from Ageia, which is free for non-commercial

purposes. The players, all the objects and even the particle systems use the physic engine to simulate a realistic game world.

The approximate raytracer is used to display transparent objects like crystals or glass. It is used for special objects in the game levels. Unfortunately, the effect is very costly, so the amount of objects using this special effect is limited.

Depth impostors are a technique to realistically render particles without clipping artefacts. They are used in all the particle systems used in the game. We could even enhance the depth impostors with another render pass to generate heat haze, which can be seen when the flamethrower is used in the game (Weapon 4).

In the next chapter, we explain how the game can be started and played. Then we give a short overview over the game engine architecture, followed by an instruction on how to create your own levels using xml files.

Finally, we would like to say thank you to Markus Giegl and Michael Wimmer for supporting us and providing us with the necessary hardware, which helped us a lot during development.

Playing the game

Welcome to the jungle!

The aim of the game is to help a poor giraffe in its nutshell vehicle to fight against some other animals in their own cars, using some of the coolest weapons you have ever encountered in gaming history! ☺

In order to play the game, your system needs to fulfil the following minimum specifications (Since we implemented some very costly effects using latest hard- and software technology, the minimum specifications are quite high):

- Windows XP/Windows Server 2003 operating system
- CPU with at least 2.5 GHz.
- 512 MB RAM
- A shader model 3 compliant graphic card (tested on a Geforce 6600GT)
- 50 MB hard disk space

The game uses version 2.4.0 of the Ageia PhysX engine. Therefore the PhysX drivers have to be installed, regardless of having an Ageia Physx hardware accelerator or not (note: it may be possible the game speed increases by using such an accelerator, but we haven't had the opportunity yet to verify this since we don't have one).
To install the driver, please execute PhysX_2.4.0_SystemSoftware.exe lying in the root directory of the game.
NOTE: If you do not install the driver, the game will CRASH IMMEDIATELY!

You can easily configure the game by editing the file config.xml. Here the game resolution as well as the display mode (windowed or full screen) can be selected. Furthermore, you have the opportunity to add your own

custom levels here by adding a “<challenge>” tag. Information on how to add a custom level will be explained in detail later.

After this installation, start the game by executing jungle.exe. The menu screen will fade in, and you can select a challenge from the list box and play it by clicking the “Load Challenge” button afterwards. If you don’t want to play (anymore), you can click the “Exit” button to leave the game.

As soon as a challenge is loaded, you see your avatar, a giraffe, sitting in its nutshell vehicle, standing somewhere on a beautiful landscape. Unfortunately, some enemies will appear and attack you sooner or later. Therefore you have to interact: steer your “car” using the well-known “wasd” buttons while aiming with your mouse! By clicking the left mouse button, the current weapon is fired.

During the game, you can collect various weapons:

1. Rocket launcher: Explosive bullets! This is the standard weapon. If you hit your enemy with a bullet, it is catapulted into the air and the terrain gets destroyed because of the impact explosion! Be careful though: if you shoot into the ground somewhere near you, your car will be damaged and your health will be reduced! Select this weapon by clicking “1” on the keyboard.
2. Bombs: They are the most destructive weapon in the game, but it is very hard to control where they explode. It creates a gigantic explosion and huge crates, and players in its area are damaged drastically! Select this weapon by clicking “2” on the keyboard.
3. Alien Weapon: The bullets from this alien weapon are not affected by gravity! Select it by clicking “3” on the keyboard.
4. Flame Thrower: Definitely a weapon only for close combat, but then very effective since aiming is easy. Be careful not to get toasted yourself while fighting in a “wheel-to-wheel”-fight! Select this weapon by clicking “4” on the keyboard.

5. Ice Thrower: You can use this weapon to freeze your opponents and make them get slower and slower, which can be very useful in combination with the Flame Thrower weapon.

Now it’s up to you to defeat the enemy animals! Be sure to pick up goodies like health packages if you find any (they appear from time to time).

By pressing the Escape button, you can return to the menu screen again, where you can load another level or leave the game.

Special Keys during the game:

- F1 – display help
- F2 – frame counter
- F3 – physx debugger on/off (physxdebug tag has to be “1” in config.xml)
- F4 – music on/off
- F9 – depth impostors on/off
- F10 – raytracer on/off
- F11 – turn off AI (enemies do not shoot anymore)
- Pause – Pause Game

The game engine

Here we will give a short overview of our game engine.

General Overview

The main class is called GameManager. It is responsible for creating the scenes (menu scene for the menu, game scene for playing the game levels) and switching between them. Moreover the game is updated and controlled from here.

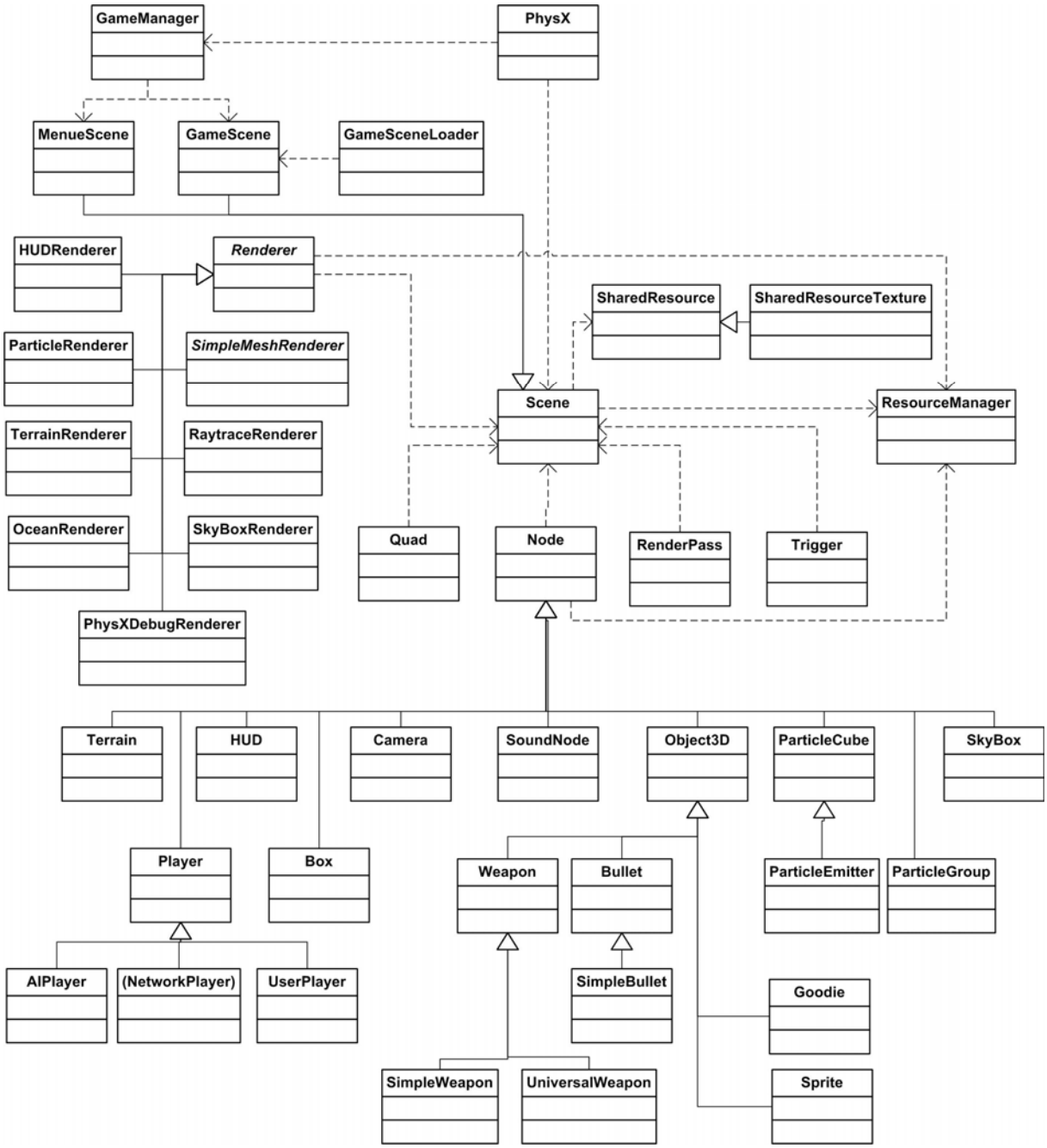
All objects in the game are derived from the class Node (for Example the class Object3d which stores a mesh and its textures). The Node class has several methods which are useful for positioning, culling and rendering each object.

Each renderable node needs to have its own instance of the (derived) Renderer class. There are several renderers for specific objects, for example the RaytraceRenderer for the approximate raytrace effect.

Furthermore, a ResourceManager class is responsible for efficient resource loading, a HUD class creates the HUD, and the Camera Class does all the camera stuff.

Of course there are several more classes which perform their own special tasks (special nodes like the Ocean or the Terrain, or all the specific renderers) which will partly be discussed later.

Classdiagram



Note: Not all dependencies are shown due to readability.

Scene graph

All engine objects are organized in a hierarchic scene graph. Each node has several functions to change the behaviour, position or rotation. The function “update”, for example, is called each frame for each node recursively. Here it is possible to do some calculations depending on the node, like updating the opponent players. Furthermore, each node can be

set to a standby mode. This means, that the node still lies in the scene graph, but no operations are performed on it. This is quite comfortable if there are objects in the levels which will appear at a later point of time.

Event Handling

We developed a time dependent trigger system for all events, except for key and mouse events. So whenever a node should get activated after some time, it can be done by setting a trigger.

There are several other types of triggers which can be set, especially for events generated by the PhysX Engine. Since it runs in its own thread, there are so called Report classes which notify our engine about certain events that occurred. These events are stored in Triggers which are immediately processed in the next frame. A typical example for a PhysX event is a bullet colliding with the terrain.

PhysX

The PhysX Engine was implemented in a way which makes it usable very abstractly. This means that every node can be used as a physic object in the scene, while the node handling does not change. To create a physical object in the engine two main steps are necessary. First the shape of the physic object has to be described. After that, we just have to tell the node how it should behave (static, kinematic or rigidbody). For more information on the different behaviour modes, please take a look at the PhysX documentation. To view what PhysX is doing during the game, open the config.xml file and set the physxdebug tag to “1”. Whenever this tag is set, it is possible to turn the PhysX debugrenderer on and off by pressing F3.

Frustum Culling

The game uses a quadtree to perform fast frustum culling. While culling the objects against the view frustum, a list of all needed renderers is set up.

Renderpasses

After frustum culling is completed, there is a list of all renderers. Each type of renderer has a priority, by which they get sorted (e.g. SkyboxRenderer renders first). Some renderers need extra passes to work correctly. In this case, these passes are rendered using the RenderPass class, which is capable of handling separate rendertargets, custom transformations and a list of functions which should be executed.

Special Nodes

Terrain

The terrain uses a heightmap to generate its geometry. It is splitted into several patches to enable culling using the quadtree. The terrain is destructible: whenever explosions occur, the vertices as well as the textur will be modified accordingly. Multitexturing and per-pixel-lighting is performed through a pixel-shader.

Ocean

To render the ocean, we created a shader which was inspired by the nvidia ocean.fx shader. It uses a normal mapping technique to generate the wave distortions.

Particle System

The particle system is a pretty customizable collection of classes to generate various particle effects. Its main parts are:

- ParticleGroup – It stores the particles and the emitters. It is possible to attach a ParticleRenderer so that all particles can be rendered at once.
- ParticleEmitter – Generates the particles. We are able to define parameters like emission duration, birthrate, velocity, rotational degree of objects, vertical and horizontal spreading and obviously which nodes should be emitted. ☺
- ParticleCube – Used to manipulate momentum and velocity of particles. It should be used as a baseclass for other ParticleCubes. Each ParticleCube has a distinct width, height and depth. Every

Particle which is inside the ParticleCube can be influenced by it. Currently there are no ParticleCubes in the scene, but for example if you would like to apply wind, tornados or similar effects to the particles, this would be the fastest way to do so.

Sound

We chose the fmod sound library to add 3d sound effects and music to our game.

GTP Effect Renderers

Depth Impostors

As previously described, depth impostors are a technique to realistically render particles without clipping artefacts. You can turn on/off distance impostors by pressing the F9 key.

Standard Billboards with Artefacts

Depth Impostors – no Artefacts

The rendering of the Particles is done by the ParticleRenderer class. But in order to render the particles, the zBuffer values are needed. Therefore this effect needs an extra renderpass. Since there can exist multiple ParticleRenderers at the same time, the zBuffer is used multiple times. To provide a suitable solution for this problem (and similar situations) a class SharedResource was developed. It is a very simple base class for some Resources which should be available in the whole scene. The derived class SharedResourceTexture stores a pointer to a texture (in this case the

result of a depthpass). Each SharedResource gets reset every frame, so that renderer can ask if the resource is up to date or not.
If a particle is drawn into the scene, a lookup into the depthTexture is made. If the pixels z-Value is near the current z-Value it is faded out. So, if the Particle hits another Object (equal z-Value), it is completely transparent.

Heat Haze

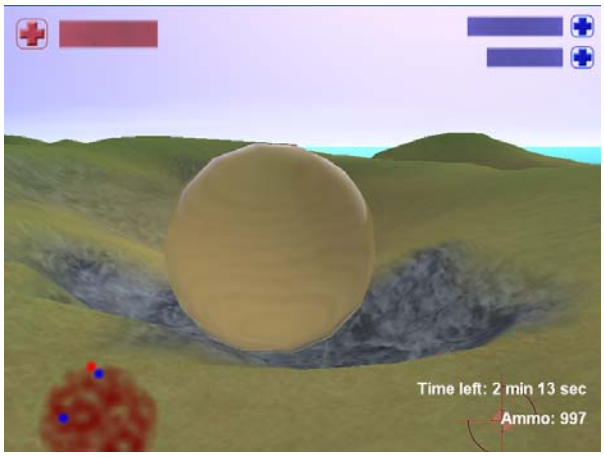
To get the effect of heat haze we extended the depth impostors. In another pass all particles which generate heat are rendered to a distortion map. Here we use depth impostors again to fade out the “influence” of heat. After that the final Image is generated by simply distorting the rendered image through the distortion map.

Approximate Raytracer

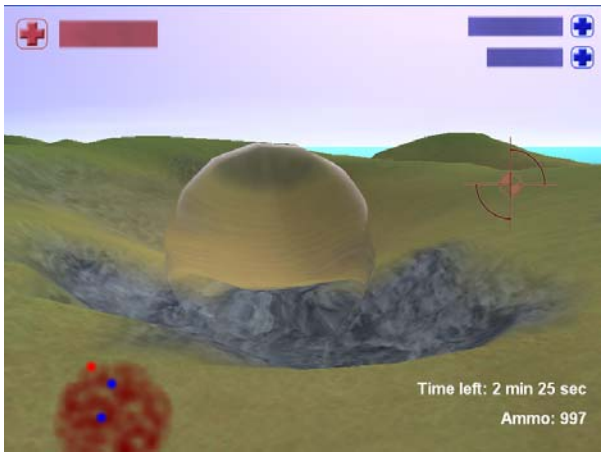
The RaytraceRenderer is used to display transparent or reflective objects. Equal to the ParticleRenderer, an extra pass is needed, but this time a cubemap of the environment from the objects position is rendered. To perform this task again the RenderPass class is used. Due to performance problems it was only possible to implement single refraction. When rendering the cubemap, the distance for each pixel is saved into the alpha channel of the texture. This is why we cannot draw any transparent objects like particles into the cubemap.
Since the algorithm works iteratively its quality strongly depends on the first derivation of the depth (the alpha channel). Sometimes (ex. crossover between terrain edge and skybox) the differences are very big, which results in artefacts. Maybe it would help to blur the depthpass before using it.
We wanted to add more objects than one, rendered by the RaytraceRenderer. Since this is normally not possible because of performance issues, we developed a small algorithm that only renders the object which is nearest to the camera. All other objects in the view frustum get updated (rerender the cubemap) about every 4 seconds. This update is done at random time intervals, which should prevent all objects

from being updated at the same time. With our solution its also possible to render inter reflections between to raytraced objects.

To change the rendering between classic refraction and the GTP effect press F10 during the game.



Classic Refraction



Approximate Raytraced Refraction

Creating your own levels

Each game level is created by specifying all resources and information in an xml file. If you create your own level (and therefore your own xml file), be sure to add a <challenge>tag in the file config.xml, which is lying in the game root directory:

```
<challenge>
  <name>Challenge name</name>
  <location>challenge_location.xml</location>
</challenge>
```

All models except the terrain are in the DirectX .x file format. To create your own level, we suggest copying an existing xml file and modifying it. Please do not change the order of the tags; it could be possible that some things stop working otherwise.

A level xml file consists of the following tags (Note: This information does not concern config.xml! A level has to be created in another xml file.):

- <challenge>: root node. This tag indicates a new jungle rumble level.

  ```
  <challenge>
    …level…
  <challenge>
  ```

- <name>: sets the challenge name.
- <mode>: sets the game mode using an integer. Currently only the deathmatch mode is supported (so please set the mode to 0).
- <time>: Specifies the time the player has to finish the level in seconds.
- <description>: Sets a message which is displayrd at the beginning of the level.
- <soundfile>: Can be used to specify a mp3 file which is played as background music while playing the level.
- <soundvolume>: Sets the volume level of the background sound.

- <sun>: this tag has a subnode <direction> which specifies the sun direction using <x>, <y> and <z> subnodes again, for example:

```
<sun>
  <direction>
    <x>-1</x><y>-1</y><z>1</z>
  </direction>
</sun>
```

- <hud>: defines the Head-Up Display in the level. The hud tag is a bit more complicated than the other tags we encountered so far, since there are many parameters to be configured:
  - <crosshair>: defines the crosshair texture. <crosshairscaleX> and <crosshairscaleY> can be used to modify its size.
  - <radar>: defines the texture for the radar which is usually displayed in the bottom left corner of the screen. To generate such a radar texture, take the terrain heightmap, reduce its size by -1 (for example, if the size of the terrain heightmap is 129x129, the texture size has to be 128x128), flip it vertically and put it into the alpha channel (since the radar has to be semitransparent). <radarscaleX> and <radarscaleY> are used to modify the displayed size, while <radaroffsetX> and <radaroffsetY> are used to define the screen position.
  - <userplayerscale> and <aiplayerscale> are used to modify the the display size of the players' health bars.
  - <texture0>: defines the "dot" which is used to display players on the radar.
  - <texture1> to <texture4>: define the healthbar textures.
- <terrain>: needs to be set AFTER the <dimension> tag! The terrain tag is used to specify the heightmap (from which the landscape is generated, defined in <heightmap>) and the textures (defined in <texture0> to <texture2>). The heightmap has to be a 129x129 greyscale raw file.
- <ocean>: The ocean shader needs three textures: <texture0> defines the normal map used to generate the wave distortions, and <texture1> sets the cubemap used to generate reflections. The

<watermap> tag gives information on the blending factor of the ocean. The <shallowColor?> and <deepColor?> tags set the water color.

- <skybox>: Sets a textured cube in .x file format as skybox. Note: The polygons of the skybox are rendered clockwise, so you can model the cube like a standard model (with the normals pointing inside-out).
- <envObject>: Defines an object which appears in the game level. Many parameters can be set here:
    - <position>: Sets the object position in the scene. Use the <x>, <y> and <z> tags to set the coordinates.
    - <rotation>: Sets the object rotation in the scene. Use the <x>, <y> and <z> tags to set the rotation around all three axes (in degree).
    - <xfile>: Defines the .x model to be loaded.
    - <renderer>: This is a tag to define a special renderer to be used for this object (optional). Currently the game has only one special renderer, the approximate raytracer (type 0).

```
<renderer>
   <type>0</type>
  <fresnel>0.0</fresnel><refraction>0.95</refraction>
   <iterations>1</iterations>
</renderer>
```

      The parameters <fresnel> and <refraction> define the appearance of the effect.

      The parameter <iterations> defines how many iterations should be calculated for each pixel. (higher leads to better results, but is slower)

    - <arrivaltime>: Sets the time when the object appears in the game (optional).
    - <timevariation>: variates the arrivaltime (optional).
    - <pmaterial>: Defines the physical material of the object.

```
<pmaterial>
  <materialid>1<materialid>
```

```
  <restitution>0.7</restitution>
  <staticfriction>0.5</staticfriction>
  <dynamicfriction>0.5</dynamicfriction>
</pmaterial>
```

<materialid> sets an id for the created material (id must be an integer > 0). This is necessary to refer to a material in the <pactor> tag. <restitution> sets the restitution, a measure of the elasticity of the collision between objects, and the two other parameters define the friction.

- <pactor>: Defines the physical behaviour of the object.

```
<pactor>
  <behaveas>0</behaveas>
  <colmode>1</colmode>
  <density>10.2</density>
  <material>index</material>
</pactor>
```

<behaveas> defines the behaviour of the object (0 - normal, 1 - static, 2 - kinematic, 3 – rigidbody), <colmode> the collision mode (0 - mesh, 1 - heightfield, 2 - convex, 3 – pmap). <density> sets the density, and <material> sets a material you have created before (!) using <pmaterial> (use 0 for a standard material).

- <sprite>: Defines a sprite-particle which can be emitted by a particle emitter.
  - <dimension>: Sets the sprite dimension.
  - <texture0>: defines the texture image file.
  - <lookAtCamera>: (0 or 1) defines if the sprite is rotated towards the camera or not.
  - <animation>: activates sprite animation. If you want to animate a sprite, you have to use a texture which consists of several single frame images. For example, you can have a texture file which has 64 frame images in it (8 rows, 8 columns). The <animation> tag has several sub tags:
    - <fps> defines the animation speed (frames per second)
    - <framecount> defines the number of frames to be used

        - <rowcount> defines the number of rows
        - <columncount> defines the number of columns
        - <loopanimation> (0 or 1). If the value is 1, the animation is played endlessly.
    - <key_start>, <key_middle> and <key_end> define animation key frames. You can set these key frames to change the particle size & color during the animation.
        - <red>, <green>, <blue> and <alpha> set the color.
        - <width> and <height> set the size.
        - <time> only affects the <key_middle> and defines when the <key_middle> state should occur (value between 0 and 1).
    - <pactor> defines the physical behaviour of the object. This tag has already been explained in detail at the <envObject> tag.
- <particleEmitter>: Defines a particle emitter. To define the particles which should be emitted, include <sprite> and/or <envObject> sub tags.
    - <position> and <rotation> are to be used in the same way as in the tags described before.
    - <dimension> sets the area size from which particles are emitted.
    - <velocity> defines the particle speed.
    - <duration> defines the time the emitter is active.
    - <heathaze> (0 or 1) activates the heat haze effect for the particles.
    - <timetolive> sets the time a particle exists.
    - <horizontalDegree> and <verticalDegree> set the emission angle for the particles.
    - <rotationalDegree> defines the particle rotation.
    - <birthRate> defines the number of particles to be emitted per second.

    - <colGroup> defines the kind of collision group the particles belong to. Choose between 4 (other), 20 (no collision), 25 (no self-collision) or 30 (obstacle).
    - <sprite> or <envObject>: Use as many of them as you want to define the emitted particles (see detailed explanation above).
- <player> defines a player
    - <type> (0 or 1) defines whether this player is the human player (0) or a player controlled by the computer (1)
    - <xfile> sets the x model to be used.
    - <position> sets the player initial position. The tag has been explained already, see above.
    - <team> define a team the player belongs to. Players in the same team don't shoot each other.
    - <weapon> defines a weapon the player possesses.
        - <type> defines the kind of weapon.
        - <amount> sets the initial amount of ammunition for this weapon.
- <goodies> defines goodies which appear in a level. There are several types of goodies, but all of them have the <position> (has already been explained several times above), the <arrivaltime> and the <timevariation> tag (see <envObject> for an explanation). Moreover, the already-known <pactor> tag as well as the <renderer> tag can be set as well.
    - <healthpackage> sets a health package.
        - <healthamount> sets the amount of health the package gives when picked up.
    - <weaponpackage> sets a weapon to be picked up.
        - <type> defines the type of weapon.
        - <amount> sets the amount of ammunition the picked up weapon has.
    - <amopackage>
        - <type> defines the type of weapon the ammunition is for.
        - <amount> sets the amount of ammunition