



## Simulation Engines Group 2

---

By: Jonathan Johansson, Peter Lundberg, Alexander Nordh, Joakim Nordh and Anton Perc

## **Abstract**

This project has been about making a simulation engine based on the graphics engine "Ogre3D" by extending its functionality with extensions made by the group. These consists of a multimedia extension implementing dynamic echoes, a physics extension implementing weapon physics, such as grenades and bullets, an AI extension implementing the path-finding algorithm A\* and a graphics extension implementing cartoon styled weather effects. This was in the end used in a demonstration application, which there is screenshots of in the report. This demonstration application is not a game as we had hoped it to be, but it shows the functionality of each extension very well.

## Table of Contents

Abstract .....	2
Introduction.....	4
Purpose.....	4
Goals .....	4
Pre-study .....	4
Specification of demands .....	5
Physics - Functional demands .....	5
Physics - Non-functional demands .....	5
Multimedia - Functional demands .....	5
Multimedia - Non-functional demands .....	5
Graphics - Functional demands .....	6
Graphics - Non-functional demands.....	6
AI - Functional demands.....	6
AI - Non-functional demands .....	6
Conceptual model and Architecture .....	7
Design .....	8
Multimedia .....	8
Graphics.....	8
AI.....	9
Physics .....	9
Classes: .....	9
Interaction: .....	10
Implementation.....	10
Physics .....	10
Multimedia .....	12
AI.....	12
Graphics.....	12
Integration and testing.....	13
Tech demo .....	14
Description .....	14
Design .....	14
Implementation.....	14
Results .....	15
Screenshots .....	15
Performance .....	16
Conclusions.....	16
List of sources.....	17
Ogre3D.....	17
Multimedia .....	17
AI.....	17
Physics .....	17

## Introduction

### Purpose

The purpose behind this project was to familiarize ourselves with the OGRE3D graphics engine, as well as with libraries and components that are used in unison with OGRE. The idea was not only to build a finished product, but to learn how to cooperate and work together, so that we would be able to present a completed project in the end. Our idea was to add meaningful extensions to an already existing framework that might have future use for other developers, or present as a starting point for further development if we decided to do so. The purpose of the engine itself is to be a way of simplifying the developers work with game making, but it could probably be usable in other areas as well.

### Goals

The main goal for the project was to make individual extensions, and then utilize them in the construction of a simulation engine. The idea was to make use of current technologies in the development. With the creation of a game engine that works well with the extensions, and a demonstration on how it works, the goal will have been reached. However, it is still possible to put more work into making the extensions better, and adding more functionality to them.

### Pre-study

The first thing that everyone had to do was to find an extension to work with. This meant searching and looking into what possibilities there were in the extension area. When an extension idea had been found, we all had to specify more precisely what we wanted to do, and then look into how we would like to implement our ideas. For many of us, this meant that we would also need to find external libraries to make the programming simpler for us, such as audio and physics libraries.

When we had decided what external libraries to use and knew what extensions to make, we had to learn more about the programming language which we were going to use, since many of us had not used it much at all before this project. After going through several "C++" tutorials, the next thing that we had to study was the "Ogre3D" framework itself. When we had learnt more about "Ogre3D" and "C++", we could start working with both engine and language and testing to see how everything worked.

We also had to get a better understanding of how our programming environment worked. For this project we worked with "Visual studio". The most important things we needed to learn about "Visual studio" was how to import the external libraries that we needed, but settings such as code formatting, indentation and code font size was good to know as well.

Another tool that we needed to learn how to use was "SVN", which we were not completely used to working with.

## Specification of demands

### Physics - Functional demands

The idea was, to create an interface. That is able to create several types of projectiles and take care of the interaction of them with the environment. Several types of different projectiles are required, as well as different interactions with the world them.

The main types are the grenade, shell and the bullet. While grenades and shells should be able to cause explosions, bullets should just return a hit. The extension should have the possibility to manage more launchers, and handle a larger amount of projectiles in the area.

### Physics - Non-functional demands

The extension should be as simple to use as possible. It should require minimum interaction beyond the basic commands, yet still be possible to be modified and expanded when the need would arise. As the amount of launchers increases the extension should be able to manage everything on it's own. The programmer shouldn't be bothered too much by searching for the correct launcher, but rather just call the one, with an ID.

The code should be well commented, making it easy to look through, as well as the functions should be clearly defined and commented, making sure that their function is clear. However, understanding how the code works should not be required for the use of the extension. Basic knowledge of how NxOgre works and a short explanation is all that would be required for a quick start.

### Multimedia - Functional demands

For my sound extension, there was a demand for it to be able to play a sound or music. It also had to be able to play a sound with an echo following it. These were the main requirements, but it was also required to be easily controlled within the game code. It should also be possible to move the listener and the sound sources in the room during the game. Another thing that must be possible is to add an delay before the sound is played, so that grenade sounds can be played when they explode. It should also be possible to scale the distance and reflection values of the objects in the sound source's surroundings within the game code.

### Multimedia - Non-functional demands

The echo sound extension should be usable by any "C++" 3D programming environment. The code should be structured into "hpp" and "cpp" files and be written in an object oriented way. The sound player should be easy to instantiate and use by the programmers, and it should be possible for other developers to make their own sound player using the "Echo" and "EchoProperties" class independently from the "Soundplayer" class.

## Graphics - Functional demands

The functional demands for the cartoon weather effects were defined as follows:

- The extension should be able to display cartoon-styled sun, moon, clouds and stars on a sky that changes depending on the time of day.
- It should be possible to add smiley faces to the sun, moon and clouds, which rotate so that they always face the camera in a reasonable way.
- It should be possible to add cartoon-styled snow and rain to a scene, with the possibility of changing velocity and density at runtime.
- It should be possible to add a wind vector which affects rain and snow particles.

## Graphics - Non-functional demands

- The non-functional demands for the cartoon weather effects were defined as follows:
- The extension should be written in an object-oriented manner, consisting of well-structured classes; no copy-pasted code, or spaghetti code that is.
- It should be well-commented.
- It should be as self-contained as possible.

## AI - Functional demands

The AI extension should be able to calculate the shortest weighted path between two nodes on a graph. Also the extension should be able to create a general graph map that is usable with it.

## AI - Non-functional demands

The A\* extension should be an independent extension that do not depend on any external libraries, frameworks or wrappers to do its calculations.

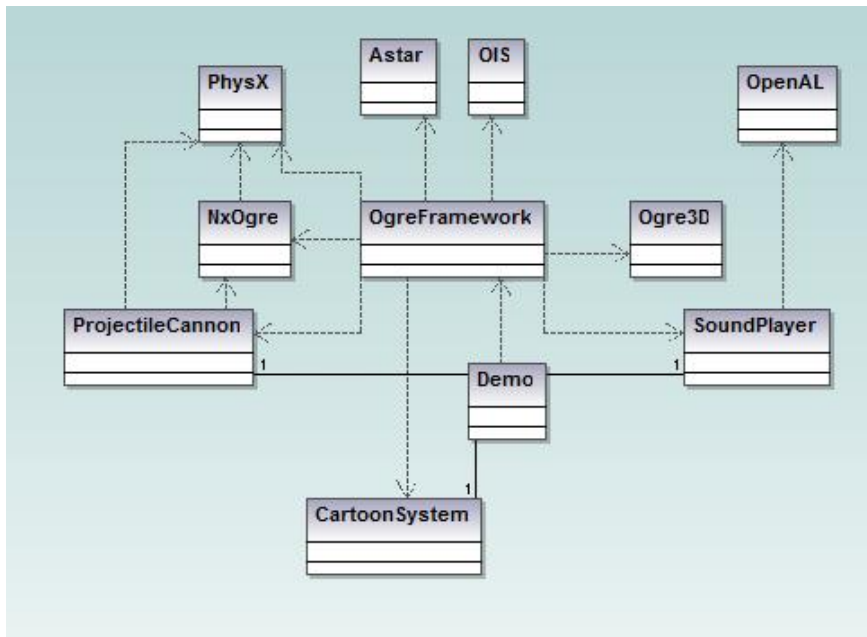
Also should the A\* code be structured in a object oriented manner to make the code easier readable for others so the code can be reused in a later stage outside the project.

The extension should be fully functional outside Ogre and be able to calculate the shortest path for any graph map as long as it is in the same format as used by the algorithm.

The code should be structured with get and set methods for every variable that is used outside of the class also there should be as few public variables or methods as possible. No global variables are allowed in my extension and to make the code easier to follow the necessary variables should be passed around to the method that needs them instead of making them public.

## Conceptual model and Architecture

The concept of our model is quite simple. Everything is in one way or another connected to the OgreFramework. The application, in our case the demo, has a few components and can reach all other functionality through the framework. The sound player is one of the components that the demo needs to have if it wants to be able to play different sound and music. The demo also needs to have a ProjectileCannon so we in the demo can spawn different kind of shells. Lastly it has a cartoon system added to it that takes care of all the weather related graphics. All these components can be skipped and the methods in their classes can still be reached since they are all parts of the framework, but without an instance of for example the sound player, the added sound wouldn't be stored somewhere and thus they can't be played.



Except for the components that the demo actually has, the rest is reached by simply having the framework. The framework takes care of all the Ogre3D related graphics and makes sure the screen gets updated. It handles the physics through a wrapper called NxOgre and that can be used to simulate all kind of physics. It takes care of input from the mouse and keyboard by using OIS and it can calculate the least expensive path between two points with the A\* algorithm.

The software flow is also quite straight forward. When the demo starts it first creates a framework that is being used for the rest of the time. Then it does a setup of the entire scene by initializing the physics, the cartoon system, the sound player and the A\*. In this setup all the physical objects, including the character you move, is also implemented and light sources are added.

After the setup and initialization step the main loop starts. The loop starts by checking if there is any new event from the mouse or keyboard and acts accordingly. It calls the A\* to calculate a path if it is currently needed. After that it handles all the game logics and physics by checking for collisions and calculating new velocities and directions. It also updates the location of the sound player to the position of the character so that all sounds will be heard as if you are at your current position. Lastly it updates all the graphics and the statistics for it.

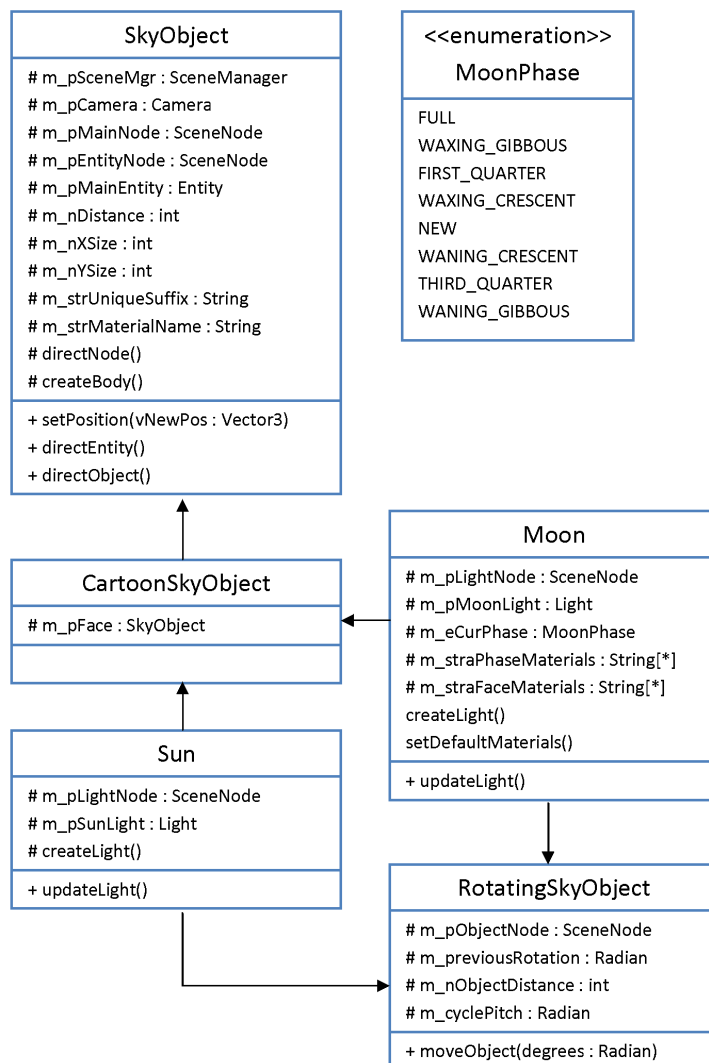
## Design

### Multimedia

The class that is instantiated in the game code is the "Soundplayer" class, which handles and plays all sounds and music whenever the programmer tells it to do so within the game code. The "Soundplayer" class can also play a sound with an echo, and then takes use of the "Echo" class. This class is made for calculating the echo's properties. These properties are its volume, its delay before playing and if it should be played at all. After doing its calculations, the "Echo" class returns an "EchoProperties" object to the "Soundplayer" class. The "EchoProperties" class is only made for being instantiated to hold these properties. The "Soundplayer" will then use the values it can get from the "EchoProperties" object to create the echo sound after waiting a certain time. The waiting is done in a separate thread so that the program can still run as normal without forcing the game to freeze. There is also a possibility to play a sound with an echo after waiting a certain time, this is used for grenades for example.

### Graphics

The primary class of this extension is the CartoonSystem class, which takes the Root, SceneManager and Camera objects in its constructor. It handles all objects in the sky, the sky itself and the particle systems which create rain and snow. To make it update its elements automatically each frame, it has to be added to the root as a FrameListener, which is an Ogre class that it extends. The class has public methods for adding and removing snow, rain and wind, and changing their velocity and density.





For creating objects in the sky, there is the SkyObject class, which creates a plane at a certain distance from the origin, and fits a material to that plane. The CartoonSkyObject extends the SkyObject class, and adds a face to the object, which itself is also a SkyObject. The Sun and Moon classes extend the CartoonSkyObject class, as well as a RotatingSkyObject class, which adds a method for rotating the object around the origin.

The CartoonSystem class creates a Sun and Moon object in its constructor, but for it to display rain or snow one has to call the addSnow()- and addRain()-methods. If a wind vector is added with the addWindVector()-method the CartoonSystem also creates a cloud with a face consisting of a smiley blowing wind from its mouth. The cloud is positioned at the far end of the world, in the direction where the wind is coming from. It will also automatically add an affector to the particle systems in place, so that the wind will affect the snow and rain.

## AI

The A\* extension contains two classes the primary class that does all the calculations called Astar and a class called AstarNode that are the objects the algorithm traverse over when it tries to find the shortest path. The AstarNodes are bound together by letting each node has pointers to the real objects of all its neighbors this way you can always traverse over every node as long as a node is connected to the graph map. The AstarNodes contains several boolean values to speed up calculations so the algorithm can just check if the current object has a specific value instead of iterating over a collection of items that has that currently has that attribute.

In the primary class I have tried to divide all repeated code parts into private methods to let them be reusable instead of having redundancy in the code. This also makes the code easier to gain insight in for people that did not write it. The primary class consists of several important public methods, these are GenerateAstarPath, GenerateGraphMap and the two convert functions specifically done to ease the communication with Ogre. The GenerateAstarPath takes a start node and an end node together with the graph it should traverse over and calculates the shortest path between them. GenerateGraphMap takes an int value as input and creates and return an NxN graph where N was the input value.

## Physics

### Classes:

The core of the physics extension is a single class. However it is not meant to be invoked each time, but more as a single instance of this specific class that will be able to manage everything related to the extension. This makes the managing easier, and it frees the programmer from managing the allocation.

There is one other class, which is located in the same file. The purpose of that class however is different. It is meant to be strictly as a contact report class. The way PhysX handles custom callbacks, they require a class that inherits the NxUserContactReport properties. This class is created in the constructor of the ProjectileCannon class.

### Interaction:

To create a new manager we just call

```
ProjectileCannon cannon = new ProjectileCannon(mRenderSystem);
```

```
int ID = cannon->addLauncher(direction, position);
```

After this, one more step is required before we can actually create the explosions in our scene. We need to call the `purge()` method. We do this by placing the `purge` method in our main game loop:

```
cannon->purge(time) //time since last frame in ms
```

After this has been done, the extension should be ready to interact with. We might want to note, that creating a launcher is not necessary, but without any, we won't be able to see anything happen.

We have to remember the ID however, this will be required for any further interaction with the specific cannon. As of the moment, the ID can't be retrieved. The class is constructed in a way, that we must pass the ID of the launcher we'd like to use each time. This is to minimize the amount of data that the programmer must store, instead the extensions takes care of storing and retrieving all data.

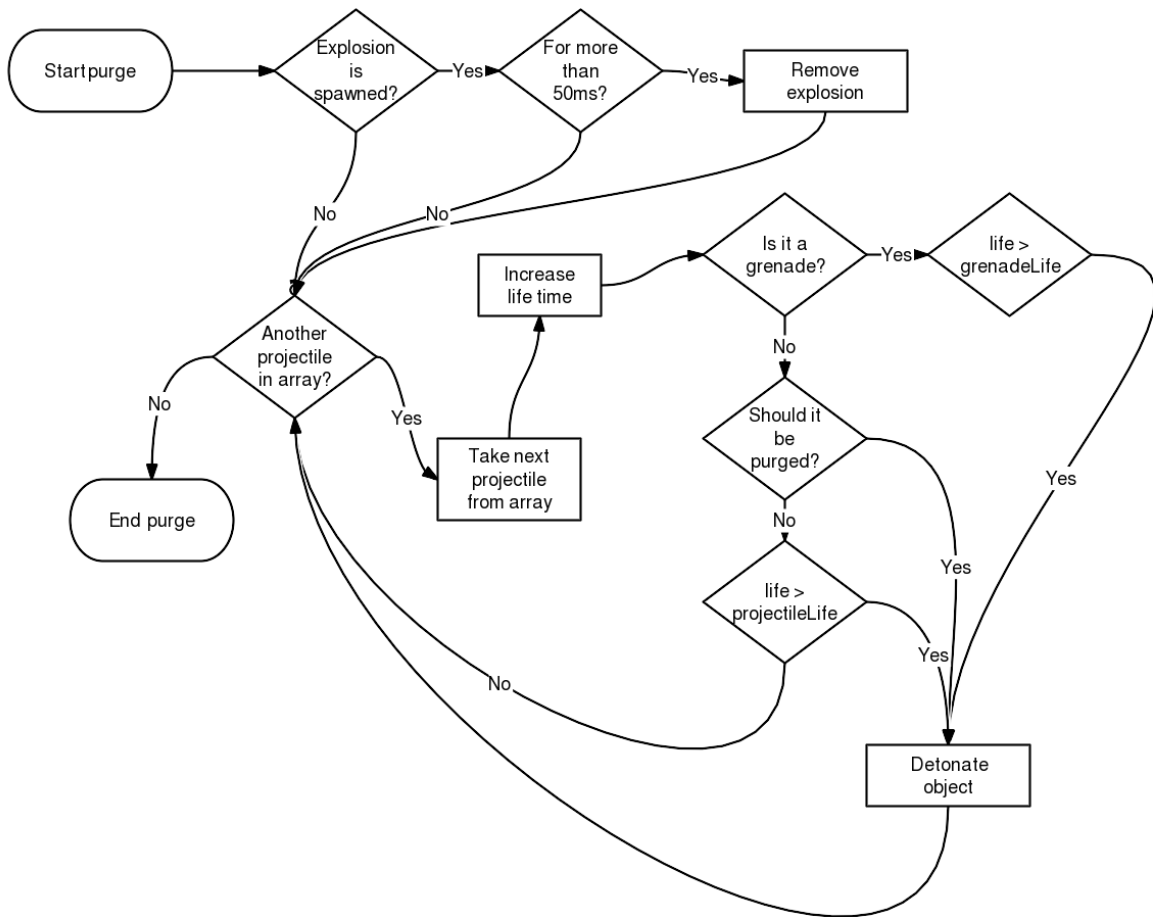
## Implementation

### Physics

The physics implementation is mostly based around using PhysX and NxOgre. The core idea was to create objects that interact with the world. This was the reason for a custom user callback class that would handle the collisions and update the data that would later be required. However, the way custom callbacks work, makes it impossible to remove any objects inside that specific class.

This is where the need for a method that constantly checks for projectiles that need to be detonated comes into play. This method constantly checks for any projectiles that might need an explosion spawned at their location, but more importantly, also removes the objects that were spawned. For example, the `fireShell` method creates an object and adds force towards a certain direction. This represents a projectile. However, giving one both an OGRE entity and an NxActor insures that we can't remove it in our custom callbacks. This is true for shells and grenades.

FastShells and bullets have a different behavior. They have no physical entity to represent them in the world. They are just points of impact that are calculated with a ray cast. Depending on the type of the projectile however, we decided whether we want to create an explosion at the contact point, or just return the value of the hit in cases of bullets.



Flowchart of the purge method.

The purge method is the core of projectiles that have a physical representation in the world, in our case shells and grenades. First we check if any explosions are spawned, and if they have been spawned for more than 50ms, we remove it, if that is the correct case.

Second part of the method focuses on going through all the projectiles currently fired, they're stored in the `pList` vector, elements of the vector are of the projectile type. We loop through all of the elements and check the attributes assigned to each of the element. First we increase the current lifetime of the current element. This is the reason why we need to continuously call the purge method. Afterwards we check whether it's a grenade, and if it is, we check if it's existed long enough to be detonated. The `grenadeLife` variable can be set depending on the user. If it's not a grenade, we check whether it should be purged. The `purged` variable indicates that there was a collision detected. The variable is set in the custom callback class. Lastly we check if the projectile has exceeded the designated projectile life. The reason for this is actually very simple. If we accidentally shoot it out of the scene, so that the projectile falls into infinity, we don't want to clog up the memory with projectiles that have no effect on the area.

After we check all the projectiles, we exit the loop and end the method. It is called again on the next game loop.

## Multimedia

The dynamic echo extension implementation has been implemented within an "Echo" class, which is used by a "Soundplayer" class. This class can handle and play different sounds and music in a 3D environment with echoes by using the "Echo" class' "calculateEcho" method, which can be called without instantiating the "Echo" class.

To be able to play sounds in a 3D environment at all, I needed to integrate OpenAL into our "Ogre3D" project. It must then keep its positions synchronized with our 3D world by setting its values to the game's values in the game's update method.

When I was going to make the delay time before playing the echo sound, I needed to find a way to make the game wait without freezing. First I tried to do this through creating a new thread by using the "Boost" libraries. When this didn't work, I used the window's API instead. This worked, and I was able to implement this solution into the "Soundplayer" methods.

## AI

The graph map is implemented as nodes connected with each other with pointers but they are also stored in a vector that contains vectors with nodes to simulate a two dimensional graph.

The A\* algorithm itself is implemented in the Astar class mainly in the method GenerateAstarPath but the heuristic method and the travelling cost are calculated in their own private methods called CalculateManhattan and CalculateCostSoFar, by implementing it in this way, it would be easy in the future to swap one of these algorithms for another algorithm if you would want to.

The A\* algorithm itself is implemented with quite a lot statement checks to check if it is valid nodes that was entered into the algorithm, if the both nodes are in the same enclosed area or if both of the nodes are walkable if the nodes does not go through this statement checks the algorithm will terminate and return an empty vector instead. With this a lot of unnecessary calculations can be skipped.

The AI extension is implemented in two parts of the demo, in the initializing where important variables are set and the first path is calculated. Also the animation variables for the robot are declared here. The other time the AI extension is called is in the demo's main loop where it moves the robot towards a node if it has a path otherwise starts to calculate a new path.

## Graphics

As earlier mentioned, all sky objects are implemented as CartoonSkyObjects, which consist of two planes with materials attached to them; one plane for its main body, and one for its face. The actual materials are made in the Ogre scripting language, and may thus be easily modified by a user without actually changing any code. For the two planes one main scene node is created, with two children scene nodes, which have the plane entities attached to them. Thus, any modification meant to be made to only one of them, should not affect the other. When created, the sky object takes a distance integer, which determines how far away from the origin it will be, although you may always set its position manually if you want to.

During each frame, the face of each sky object is rotated so that it faces the camera. This is done by creating a temporary scene node object, set right in front of the camera, with the orientation of the sky object. Then it checks for the differing angle between their normals, rotates the sky object accordingly, and removes the temporary scene node.

All RotatingSkyObjects are also rotated by a certain degree. The moveObject()-method of this class moves the object according to a preset pitch value, which is the degree which the movement is turned away from the x-axis.

The particle systems are implemented with the Ogre scripting language, using textures with transparency. Each system consists of eight emitters, which create particles at a certain rate. There is also a gravity affector, which makes all particles fall downwards. The particle systems are created right above the camera - and follow it when it moves, so that a player will always see rain and snow. Also, when a wind vector is added as an affector to the systems, they are moved in the opposite direction.

## Integration and testing

Our starting point was the graphics engine "Ogre3D", but since this was not able to do much more than just rendering objects to the screen, we needed to integrate other external libraries and our extensions to make it more useful for a game developer making a 3D game. There were four different external libraries that were integrated into our game engine. These were: "OpenAL" for sound and music, "PhysX" and "NxOgre" for physics and "OIS" for user input. This is done by including all extensions and external libraries in the framework. When the extensions are integrated with the framework, all other classes that belong to the demo application can reach them.

When all extensions had been put together into the game engine and tested, we basically had a few problems with every extension. However, we were able to solve or fix most of the bugs and problems. When testing our demonstration application, we worked as a group, trying to solve whatever problems we encountered together.

We had some version problems with the objects in NxOgre, since the objects compiled in different versions of "Visual studio", does not work in other version of "Visual studio". This was solved by letting each project member that had a diverse version of "Visual studio" compile their own version of "NxOgre".

One problem with "OpenAL" was that when adding the external library "ALut" to it, the ".DLL" files would not be placed in the correct folder automatically, and had to be placed in the windows system folder manually. This was solved by putting the ".DLL" file in the release folder instead.

For handling threads, the dynamic echo extension used the external library "Boost" at first, but when it did not work, the solution was changed to using the "WINAPI" instead. Because of this, our framework can only be used on the windows platform. The "WINAPI" was also used in the path finding implementation to create a worker thread that handles heavy calculations.

## Tech demo

### Description

The demo was created as a simulation that could show the features of all individual extensions, and what could be achieved by using them together.

### Design

We chose to have a design as simple and realistic as possible, so that the user would be able to clearly view the different extensions. You are placed in a normal outdoor environment with all the necessary components needed for trying out the extensions. We have physical boxes placed in nice formations that you can easily shoot with different projectiles to see how they react. A lot of big items are placed in the surrounding environment so that the all sound will have to take them into consideration when creating echoes. These big items, such as rocks, wells and houses, are also used to show how that the AI works. The AI is represented by a robot singing the “Mr. Roboto” song, which follows you around, avoiding all obstacles and choosing the fastest way to the location that it last saw you at. We also added a clear blue sky so that it would be easy to see the different weather effects and the sun and cloud objects on it.

To make the simulation even more realistic we decided to add shadows and different light sources. The graphics extension had already placed a light source following the sun, but to make the shadows even nicer we added an additional light source at a house so you get double shadows and some shadows at night as well.

Since first person shooter games is so normal these days we decided to also add a crosshair in the middle of the screen so that you can actually aim and get a more familiar feeling when you walk around in the world.

### Implementation

All the extensions were implemented into the demo in different ways. There is always a sound playing from the walking robot and this sound will get higher as the robot gets closer to you. We also implemented playable sounds that you can start by using different keys on the keyboard. With this, you can see how the echoes are affected depending on the surroundings. The sound is also integrated with the different projectiles, and different sounds are played for each of them. There is also a delay that makes sure that the sound from a grenade is not played until it actually explodes - as opposed to playing it when you actually throw it.

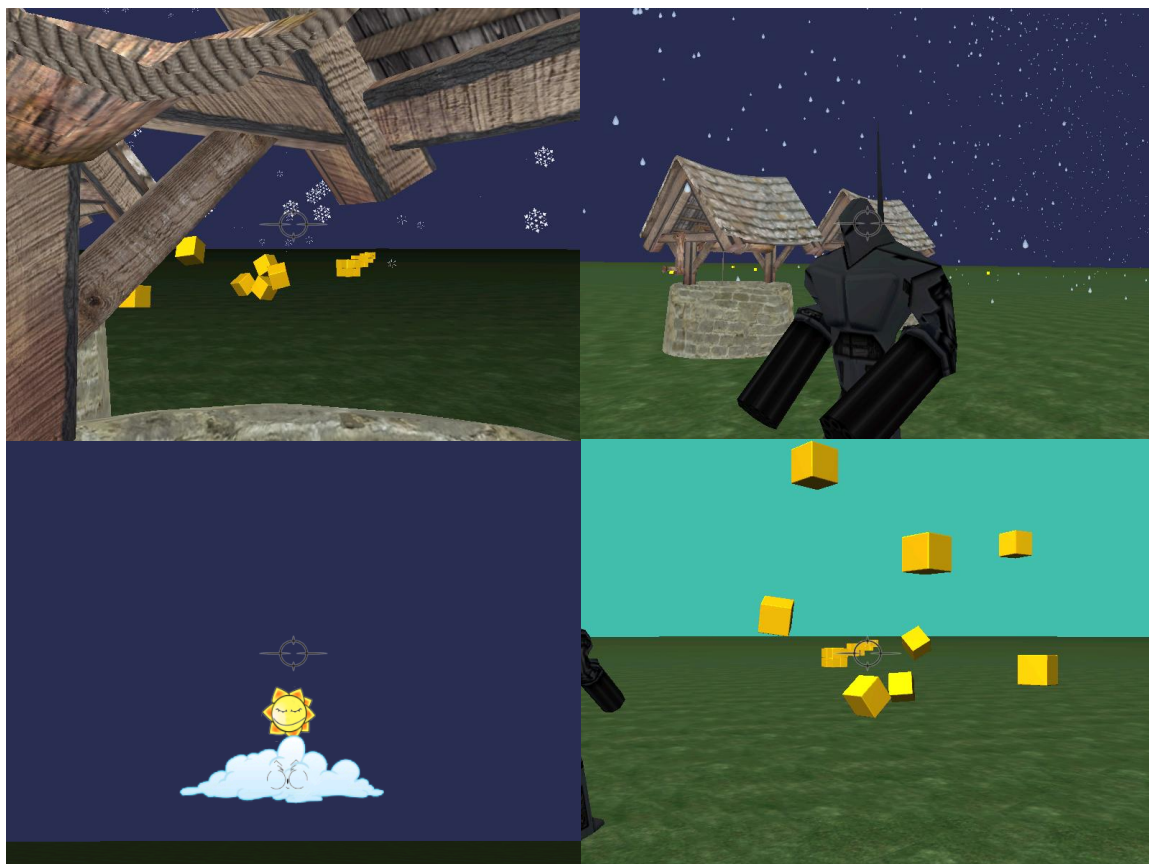
There are three different kinds of projectiles implemented in the demo: grenades, shells and fast shells. You can change between these different projectiles by pressing the numbers 1 to 3 on your keyboard. When you press space or the first mouse button, you will fire the projectile that you have currently chosen to be active. Lots of boxes have been included in the physics, which will react in different ways to the projectiles. The projectiles will also react to other solid objects, such as the rock and house. The robot uses the A\* algorithm to calculate the fastest way to you from its current location and then starts to walk towards you, while avoiding any solid obstacles in its path.

Through the graphics extension we implemented a sun that orbits the earth. When it is down, its light source is turned off, so that it will not reach the objects on the ground - thus they do not cast any shadows. A short time before the sun goes down, the face of it changes expression to a sleeping face. The skybox also changes to a darker blue when the sun goes down, representing that it is now night. Weather effects are also implemented and can be started by pressing a single key. The intensity of these weather effects may also be modified by pressing a second key.

## Results

Although it was not a very dashing demo, we are quite confident that it showed what we had done in a satisfying way, and in general our extensions worked as intended when used together. This result is shown in our demonstration application, which contains a basic environment with a sun, a cloud, a moon, weather, a robot, wells, rocks, a house and a lot of boxes. The boxes are made for showing how the physics extension works, while the sun, moon, cloud and weather is showing the weather extension. The audio extension is shown by playing sounds and hearing how the sound reflects back, together with the music following the robot. The AI extension is shown as the robot, which follows the player where ever he walks.

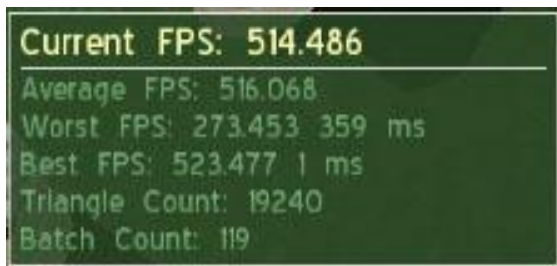
## Screenshots



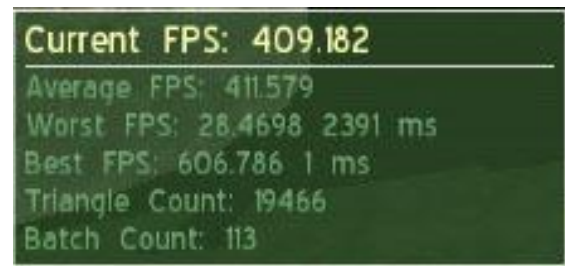
## Performance

While the performance does not cause much problems, the A\* algorithm has a glitch, and it freezes the program every now and then. This glitch occurs if the program wants a new path for the robot before the worker thread has finished executing. This is because it is the worker threads job to supply the A\* extension with a new graph map every time the robot needs a new path. The reason for recalculating the graph map is because even if a copy of the graph map is sent to the A\* algorithm, it changes the original graph map due to that the nodes got pointers to their real neighbors even if the node itself is a copy.

The performance differences between an ogre world where we have all extensions running compared to a clean world without extensions is on average around 20% slower. Which is not a huge drop compared to everything the extensions add. It still runs on average on an FPS that is much higher than the frame rate of a regular screen. A notable thing is that the worst FPS is considerably smaller with the extensions running - sometimes dropping down to below 30 FPS. Without extensions, the FPS never drops below 270. These frame drops appear mainly when the worker thread needs to create a new graph map or when new particle systems are added.



Current FPS: 514.486  
Average FPS: 516.068  
Worst FPS: 273.453 359 ms  
Best FPS: 523.477 1 ms  
Triangle Count: 19240  
Batch Count: 119



Current FPS: 409.182  
Average FPS: 411.579  
Worst FPS: 28.4698 2391 ms  
Best FPS: 606.786 1 ms  
Triangle Count: 19466  
Batch Count: 113

## Conclusions

Our goal was barely reached, but in the end, we did succeed in making a demonstration application with each of our extensions integrated into it. We could however have made it better with more functionality and better performance, but we are satisfied with that our main goal was reached.

Our demonstration application lacks a central theme, since we started out focusing more on creating great extensions rather than on making extensions that would work well together in a specific game. Because of this, we needed to make a demonstration application with what we had, instead of trying to make it into a real game.

The game engine could be used for helping a programmer in making a 3D game, and it would be even more useful if that programmer needs the more specific higher level functionality that is provided by our extensions.



## List of sources

### Ogre3D

[http://www.ogre3d.org/wiki/index.php/Ogre\\_Tutorials](http://www.ogre3d.org/wiki/index.php/Ogre_Tutorials)

<http://www.ogre3d.org/forums/>

### Multimedia

<http://www.worsleyschool.net/science/files/echo/echo.html>

[http://connect.creativelabs.com/openal/Documentation/OpenAL\\_Programmers\\_Guide.pdf](http://connect.creativelabs.com/openal/Documentation/OpenAL_Programmers_Guide.pdf)

<http://connect.creativelabs.com/openal/default.aspx>

<http://www.devmaster.net/articles/openal/>

[http://mrl.nyu.edu/~dzorin/igf06/lecture10/OpenAL\\_install.txt](http://mrl.nyu.edu/~dzorin/igf06/lecture10/OpenAL_install.txt)

### AI

<http://www.policyalmanac.org/games/aStarTutorial.htm>

<http://theory.stanford.edu/~amitp/GameProgramming/>

[http://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm](http://en.wikipedia.org/wiki/A*_search_algorithm)

### Physics

<http://www.ogre3d.org/wiki/index.php/NxOgre>

<http://www.ogre3d.org/addonforums/viewforum.php?f=6>

nVidia PhysX API documentation