Creating Serious Games by integrating external components in commercial game engines

Propositions and guidelines for future work with serious games

Niklas Henriks
Creating Serious Games by integrating external components in commercial game engines

Examensrapport inlämnad av Niklas Henriks till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information. Arbetet har handletts av Fredrik de Try (Saab) och Henrik Engström (HiS).

Submitted by Niklas Henriks to the University of Skövde as a dissertation towards the degree of B.Sc. by examination and dissertation in the School of Humanities and Informatics.

2007-01-21

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

I hereby certify that all material in this dissertation which is not my own work has been identified and that no work is included for which a degree has already been conferred on me.

Signerat:
Signature: ________________________________
Creating Serious Games by integrating external components in commercial game engines

Niklas Henriks

Abstract

The military industry has come to look at games as means for better looking and cheaper simulations. Altering games and game engines are not necessarily easy and finding a suitable engine is essential to project success. The features and design of an engine must to a large degree overlap that of the project design. Creating ‘systems of systems’ by integrating external components/systems with games is what to a large extent differentiate military serious games from other fields. However, this is not an easy task, as games are not designed with interoperability in mind. This report explain how games and game engines can be used to create military serious games, and by that explain what need to be done to have a game interoperate with external systems, how to interact with the game engine, and give guidelines to the process of evaluating and selecting a game engine. The report also argues that game engines are not always the optimal solution.

Keywords: Military, Serious Games, Evaluation, Game Engines, Integration.
Table of Contents

1 Introduction ................................................................................................. 1
  1.1 Report organization .................................................................................. 2

2 Background to Games and Simulations ...................................................... 3
  2.1 Serious Games .......................................................................................... 3
    2.1.1 Definition of Serious Games .............................................................. 3
    2.1.2 Serious games and external integration .............................................. 4
  2.2 Components in game engines .................................................................. 6
    2.2.1 Graphics ............................................................................................ 7
    2.2.2 Physics ............................................................................................... 8
    2.2.3 AI, Artificial Intelligence ................................................................. 8
    2.2.4 Sound ................................................................................................. 8
    2.2.5 Network ............................................................................................. 8
  2.3 Simulators and Games ............................................................................. 10
    2.3.1 Distribution of cost ........................................................................... 10
    2.3.2 Hardware compatibility ..................................................................... 10
    2.3.3 Creating the game/simulation world ............................................... 10
    2.3.4 Technical aspects ............................................................................. 11
    2.3.5 Vega Prime ....................................................................................... 11
    2.3.6 Unreal Engine ................................................................................... 12

3 Problem description ..................................................................................... 14
  3.1 Objectives ................................................................................................ 14
  3.2 Targeted readers ..................................................................................... 15
  3.3 Method ..................................................................................................... 15

4 Realization ................................................................................................... 16
  4.1 Evaluation Criteria .................................................................................. 16
    4.1.1 Using the game as it is ..................................................................... 17
    4.1.2 Moding ............................................................................................. 17
    4.1.3 Licensing a game engine ................................................................... 17
    4.1.4 Starting from scratch ....................................................................... 18
  4.2 Integration of external component ........................................................... 19
    4.2.1 Domain background ........................................................................ 19
    4.2.2 Battlefield 2 game engine ................................................................. 19
4.2.3 Simnon a simulator of ship dynamics ............................................... 19
4.2.4 Legacy systems .................................................................................. 20
4.2.5 Interfaces do they last forever ............................................................ 21
4.2.6 Documentation ................................................................................... 21
4.2.7 Support ............................................................................................... 21
4.2.8 Flexible components ......................................................................... 22
4.2.9 The distribution of power ................................................................... 22
4.3 Implementation ....................................................................................... 24
4.3.1 A distributed interface ........................................................................ 24
4.3.2 SPN – SeamanPhysicsNode ............................................................... 25
4.3.3 SI – SimnonInterface ......................................................................... 25
4.3.4 SCI – SimnonClientInterface ............................................................ 27
4.3.5 WinMain ............................................................................................ 28
4.3.6 ServerProcess .................................................................................... 29
4.3.7 netTypes.h ....................................................................................... 32
4.3.8 SnailNet ............................................................................................ 32
4.3.9 SimnonCommunicator ....................................................................... 32

5 Conclusions .............................................................................................. 35
5.1 Summary ............................................................................................... 35
5.2 Discussion ............................................................................................. 36
5.3 Future work .......................................................................................... 37
1 Introduction

The military industry have traditionally evaluated, visualized and integrated their systems in strict military simulations and for a long time these kinds of simulations were the cutting edge of graphical representations, but this is no longer the case. According to Lewis and Jacobson (2002) and others, 3D games and game engines have over the last few years surpassed traditional simulations and the gap is only getting wider.

Saab as well as numerous of other companies and actors worldwide are now looking towards games to gain use of their graphical superiority. All over the world there are initial studies being made to see if there is any way of making good use of games in different types and genres. Most of these studies come to the conclusion that there are ways of making good use of games for other means than just home entertainment.

The areas strongest represented on the field are schools (education), military (training, system evaluation) and hospital/medical research (training, distraction, patient education). Even though there is no doubt that there are good ways of making use of both the good graphics and the abilities for enhanced learning in games some actors tend to give a somewhat glorified view of the process. As read in Holmquist (2004)

“Off-the-shelf video games can be modified for military use for less than $200”

He gives the example of one such implementation where the game Spearhead was evaluated by the US Army Research Institute. According to Holmquist (2004)

“…game was linked via a third computer to communication and command software Force XXI Battle Command, Brigade-and-Below (FBCB2). All events that took place in the video game appeared on the FBCB2 system.”

What is not mentioned in his article is the fact that the game Spearhead is developed by MÄK one of the largest actors in the industry for distributed military simulations, and so the compatibility and interoperability between the given systems was not representative for the game market. A price tag of $200 is most likely not a realistic price for integration between a military system and a generic “Off-the-shelf” video game.

Creating 'systems of systems' where external components have been integrated into computer games or games made to interoperate with external simulations (or other types of systems) is a part of serious games for which the military industry have shown a great interest. This report aims to evaluate to what extent games can be used to create military applications and training tools through the use of integrated external components. Integration and interoperability with and against other systems is what to a large extent differentiate the military’s uses of serious games from other fields. The problems involved with such work will be the main focus of this report and the report will also explain how such implementations can be conducted, and will give guidelines to aid the process.
1.1 Report organization

Chapter 2 will initially give the reader a short background to the concept of serious games. After that follows background to game engines and how these work. Concluding this chapter is a section about the correlation between game engines and simulators, and how these compare to each other.

Chapter 3 will explain the problem statement, the methods used for evaluating it as well as defining words and subjects.

Chapter 4 give a deeper look at the usages of integrated external components and also show how such work can be conducted. The major part of the chapter is about the implementation that was created to test the capabilities of this concept at Saab in mid to late 2006.

Chapter 5 will conclude the report with a summary, discussion, evaluation, and the identification of future work.
2 Background to Games and Simulations

People respond differently to the emotionally loaded word *game* depending on whether they did or did not play electronic games while growing up. This is basically a generation-gap issue because children who have grown up since the 1980s have been exposed to electronic games their entire lives (Zyda 2005). When speaking of a game or games this report refers to them exclusively as digital games.

There is a wide variety of definitions of digital games across the range of academic, Internet and media writing. The terminology also varies between authors and over time, and is often interchangeable. For example, the terms ‘computer game’ and ‘video game’ used to refer to PC-based games and console-based games respectively but are now used interchangeably. (Kirriemuir and McFarlane 2004). Mitchell (2004) also supported this view and also point out decreasing differences between computers and the new game-consoles.

As this report mainly target persons with a background in an industry where the work is more focused on system development and integration rather than game development, a quick background to games and game development is probably required for the general reader. This chapter will give a short background to the concept of serious games, how game engines are internally structured and how game engines relate to simulators.

2.1 Serious Games

This chapter will introduce the reader to the subject of Serious Games and what subjects we can apply in military serious games. In general Serious Games can be described as the use of games as a platform for engaging, interactive, visually appealing, learning environments.

The phrase “military serious games” is here referred to a system that in some way uses a game or games to aid a military cause. As pointed out before, the possibility of integrating games with other systems is something that the military industry are looking closely at and in such that goal is made a part in the meaning of “military serious games” even though it is not exclusively a part of every military serious game.

2.1.1 Definition of Serious Games

There are two fields in Serious Games, which are not mutually exclusive; Game(s) based learning (GBL) which try to harness the motivation power of games and ‘learning through doing’. The other field tries to use game engines, modified or not, as a cost effective way to visualize a problem domain or use them as parts of full-blown simulations. Modifications in this sentence are often based on connecting the game engine to other tools or software. An example of this is described in McGrath (2004) and McGrath (2005) where the game engine Unreal from EpicGames was connected to a simulator from Anesoft that simulates patient vital signs.

Some writers would argue that GBL exclusively is the same as Serious Games but as it is possible to have a simulation built on a game engine, without the elements of fun, that can’t be the whole truth. Likewise it is possible to have a game for learning with a serious purpose that is not a simulation and that will still be a serious game, and one can even have a cost effective simulation which harness the playful elements that enhance learning and in such combining the both to its best.
Background to Games and Simulations

There are more or less just as many different definitions of the word Serious Games as there are reports about it. Some writers just define it within their problem domain such as often read in reports about GBL, other start out with a definition of games and the use of play.

Both games in general and electronic games have been around for so long that everyone knows what they are all about. The same goes for the word “serious” but Serious Games are not the combined meaning of two words but rather a concept and have as such been given a broader meaning than simply the two words. GBL is without question the largest part of this concept especially in the field of research. Nevertheless, other means of usage of games also fit within the concept of Serious Gaming, such as the use of game engines as visualization or simulations tools. For this report Serious Games will simply be defined as:

“A game or the use of a game which has or has been given a prime meaning that is not entertainment”.

Both GBL and the use of games in military training have a history that goes back just as long as games. One example is Xiangqi or Chinese chess, which have a history that goes back to 4th century BC. Xiangqi give a good way of training planning and strategic thinking and at the same time combines the engaging aspects of a game. It is thought to originally have been used as a training tool in the armies of ancient China. (Wikipedia n.d.)

Moving on in history the first electronic games were made on equipment originally made for serious usage. The first electronic games ever were created in the early ’50s and played on an oscilloscope. The first computer game called “Spacewar!” was created 1962 by students at MIT and played on a PDP-1 (Wikipedia n.d.). One can only guess from the hits that followed that it was fun and the players did have to learn how to use the equipment, at least to some degree.

2.1.2 Serious games and external integration

In the project UnrealTriage McGrath (2004) and McGrath (2005) integrated a simulator from Anesoft that simulates patient vital signs and EpicGames’s Unreal Engine. The simulation involved several emergency response players put in scene at an airport where a small airplane carrying 30 passengers just has crashed. The players then have to locate and categorize the victims into one of four treatment categories: Red (immediate), yellow (urgent), green (delayed), or black (fatally wounded). The project used the Karma physics engine, which was a part of the Unreal distribution, for all in-game physics behavior. McGrath (2004) and McGrath (2005) use an API called Gamebots to externally interact with the game engine, as they did not license the full source code. The Anesoft vital signs simulator cannot directly feed data to the game but instead create comma-delimited files that is parsed and saved in a MySQL database (McGrath, 2005). The database is then queried every 30 seconds and the information is passed through various programs and interfaces to update the casualties vital signs inside the game.

The data from the game was also extracted and sent to an external triage casualty management system as if the information was sent from real biomedical and GPS sensors. McGrath (2005) also argues that with this data the management application could be tested and improved without the need for expensive field tests.
Background to Games and Simulations

McGrath (2004) and McGrath (2005) conclude that though the engine “has proven a quality environment for basic game design and modification”. Game engines need to address fundamental simulation interoperability as a part of their design.

“Game-based exercises, embedded training systems, and virtual prototypes of C4ISR systems will require games that ‘talk’ to each other and to other external applications”. McGrath (2005).

As stated above UnrealTriage only update the information once every 30 seconds, in many types of applications this is likely not enough. In this implementation the data need to pass several interfaces, and the more layers the information need to pass the slower it will get and if the game is to look any good the game’s update frame rate need to be kept at least 30 frames per second.
2.2 Components in game engines

In this section the report will take a closer look on the technical foundations of games and briefly describe each part. This is important, as without a firm knowledge and understanding of game engines it is impossible to make a stable integration.

Both BinSubaih et al. (2006) and McShaffry (2005) illustrate the game foundation in the form shown in figure 1 with a Game Application layer that handles hardware and operation system specific calls. A Game Logic layer manages the game world and how it changes over time. The Game view or views present the game state with graphics and sound. (McShaffry 2005).

![Diagram of game components](image)

Fig. 1. Foundation of a game proposed by BinSubaih et al. (2006) and McShaffry (2005)

In this model of a game McShaffry (2005) describe the application layer as something that, if done correctly, can be disassembled from the rest and changed without this changing anything else in the game. The Game application layer handles all hardware and operation system-specific calls and provides an abstract high-level interaction against things as threads, memory and input/output for the game. Game logic is explained as the part that makes up the game, here one find the game world and all aspects and modules that defines it. One such module is the physics, which closely controls all object dynamics. The Game views are essentially the interface for the game, both in and out. Input from the player, both local and networked pass a game view, which translates the event to game code. Events in the Game logic are passed to other game views giving the player output such as the rendered image on the screen and the sound in the speakers (McShaffry 2005). As a comment to this it can be said that the renderer (graphical engine) can be found as a part of a Game view.
Lewis and Jacobson (2002) provide another picture shown in figure 2 where the layers are split into smaller parts in a more direct way. They define the engine as a collection of modules that do not directly specify the game’s behavior (game logic) or the game environment (level data). Lewis and Jacobson (2002) include in the engine the modules that handle input, output (such as rendering to screen, music and other feedback) and physics and collision detection/handling.

Further they explain the Virtual World as the topmost levels of the game, this can be scenarios or specific maps. Using graphics supplied by the game, and/or made by other players, totally new maps and worlds can be created. Editing this level of the game can usually be done through editors that are included with the game and no programming skills are required.

They define the Game Code as a layer that handles most of the basic mechanics of the game. This can be simple physics, display parameters, networking and animation control. Changing things at this level are usually done through a game-specific scripting language and require a good knowledge of the internal functions of the game. (Lewis and Jacobson 2002). The Network Code supports a robust, built-in protocol similar to DIS/HLA, which let other players interact in the same virtual environment (Lewis and Jacobson 2002).

If the game support the model stated above it is also likely that the game’s original GUI can be changed using scripts. Some new games also provide the means of changing the GUI through XML and small scripts saved to files. Two examples that use this method are World of Warcraft™ from Blizzard Entertainment and Sid Meiers Civilization™ IV from Firaxis Games.

Even though BinSubaih et al. (2006), McShaffry (2005) and Lewis and Jacobson (2002) propose two different ways of organizing the game both models include all aspects needed for a game with a reusable game engine. These essential aspects are described in the following paragraphs.

2.2.1 Graphics

The first thing most people likely think of when talking about a digital game is the visual aspects. Everything can be seen on the screen is there due to the fact that the game has a renderer or graphical engine. As the renderer is the deepest core of any game engine, making changes to it is seldom recommended and it is not likely that it is possible to do so unless the full source code is at hand.
2.2.2 Physics

Physics engines calculate as the name implies the physics of everything in the game. If a rock in the game have a mass and there is a gravity set for the world, the position of the rock will be calculated and changed with respect to the formula used. Another vital part usually included in the physics engine is the collision detection and handling. Collision detection is the process of checking objects against each other and testing if they have collided. Collision handling is when the game knows that a collision has occurred and it does something about it. Just knowing that the rock hit the ground will not make it stop or bounce back up. The game therefore needs some way of dealing with the collisions in a nice manner according to the physics of the game world. The parameters and behavior of the game worlds physics is often changed between games unless they all strive to be realistic copies of the real world. Many games have intentionally unrealistic physics to enhance the fun factor, especially when it comes to vehicle physics. The physics is something that can often be changed and manipulated both through easy parameters, such as weight of objects, or through the use of scripts. It is not uncommon that game engines use a third party developed physics engine such as the highly rated Havok engine by Havok.com, Inc. Havok has been used in some of the most successful games ever such as Halo 2, Half-life 2 and more than 150 others. (Havok, n.d.)

2.2.3 AI, Artificial Intelligence

If a game is not a strict multiplayer game or for some other reason without computer controlled entities there is a pressing need for some form of AI (Artificial Intelligence). It can be something as simple as intelligent dialogues or more advanced types as combatants in a FPS game. Later FPS games make good use of their advanced AI as a nice feature in the game, the enemy can’t simply withstand more bullets than the player but instead they use obstacles as cover and make good use of grenades and mines.\(^1\)

2.2.4 Sound

Sound engines mix and play the environmental effects and musical sounds of the game. Having good sound in a game is just as essential as having good sound in a movie. Most game tries to have their sound as close to reality as possibly but just as in movies things get enhanced to give a larger experience. In a simulation a correct sound is there to enhance the reality of the situation and not so much for a more interesting experience. Sound engines such as the extensively used FMOD are also a component often licensed from third party developers. FMOD is used in many games and the most well know would be World of Warcraft. (Firelight Technologies, Pty, Ltd, n.d.)

2.2.5 Network

Networking has become more and more essential to game as the Internet gets faster and more information can be transmitted in less time. There are good third party network libraries to be used but still most developer choose to make their own network code as it tends to get very specific for each game. Exceptions to this are the highly complex network engines used in MMOs (Massively Multiplayer Online).

---

\(^1\) Good examples if this can be seen in Halo 1 & 2 from Bungie Studios and in F.E.A.R. from Monolith Productions.
Background to Games and Simulations

where up to 100,000 players connect and play in a shared world. Creating such an environment take a lot of skill and time and therefore more than one actor in the game industry have licensed their network code from third party developers.

If putting all this components together one get a game engine or a simulation depending on how it is used. The largest technical difference between a game and a simulation, on a technical level, will be found in the physics representation. A game will most often settle with “good enough” whereas a simulation will need to have a much higher degree of correctness. One thing many times easily overlooked is the fact that no matter how advanced the engine is things will still not look good if the textures and art are not well made.

An ‘external component’ is here seen as a program/application or any other system or part of a system that is not a part of the original game or created for this specific game. An example if this can be a radar station that uses a game to either show real data or it is fed with extracted data from the game and show that on its normal display.
2.3 Simulators and Games

For simulators integration of external components is nothing strange or new, it is done all the time. But there are tradeoffs with simulators and some things games simply handle better.

2.3.1 Distribution of cost

More than one populistic article in warfare magazines prove their writers lack of insight when stating questions as to why they can’t have simulations with a prize tag just as low as games’. When it comes to budget one of the prime advantages that games have over simulators is their targeted audience. A good game (called an AAA title, oral; triple ‘A’) can be sold in many millions of copies. One such successful game is World of Warcraft™ (WoW) which is an MMORPG (Massively Multiplayer Online Role Playing Game) for which players pay both a price for the game in the store and an annual monthly fee of about US $15, there are currently over 8,5 million people around the world playing and paying for WoW every month. In the EU and US this is a ‘one of a kind’ game but it still shows the difference in audience very well, companies like MultiGen-Paradigm (creator of Vega Prime) simply can’t distribute the cost on such a large client market and hence the costs for each seat is larger. On the other hand simulation industry should be able to extend their business towards game industry and in that way get a broader audience and lower end user costs. As for now games are much cheaper per seat compared to an advanced simulator as Vega Prime where the cost per machine will be well over the 3% of sales value which is what EpicGames charge for their Unreal engine.

2.3.2 Hardware compatibility

As of tradition military simulations have run on specific hardware and often on expensive setups such as machines from Silicon Graphics. This made it very easy for the industry as the programs only needed to support one set of hardware. A game on the other hand should be able to run on almost any kind of hardware and hence has been tested on thousands of different combinations of various setups. When creating something that should be run on any generic pc and for a wide audience, licensing a successful pc game engine can save a lot of time and money as hardware testing has already been done for the release of the game. The word successful is a key word here as there are many games that have fallen due to lack of testing and hardware compatibility issues. When working with serious games or any other project you don’t want to search for bugs you did not create, bug testing is something you get paid to do not pay for doing. So examining the success and reputation of a game and game engine is important even in this context.

2.3.3 Creating the game/simulation world

The world seen in games are most often built in some kind of editor by a level designer who use the editor to place objects in the game world. When creating large simulations it is common to create the virtual world by loading real world data into the simulation and from this data create large world databases.

It is partly this automated generating process what makes simulations look visually inferior to games. If this generating program only has map data and a flight photo there is not much else to make for example a house look good, it will be the box that

---

2 Pricing is not the same around the globe, in China there is no monthly fee but instead a per hour cost.
Background to Games and Simulations

is known all too well. A house in a game is hand made by a graphics guru who have spent a good deal of time taking it to his heart and making it the best looking house he can, and that goes for the textures to. Unless a project has an extremely big budget and a lot of artists, the simulation will never look as good as a game and this applies even if a game engine is used. Money and time need to be spent on models and textures in order to make a simulation look as good as a game.

As graphics is such a big market there are numerous companies that only work with creating 3D models so if the budget is not a problem outsourcing as much of the work as possible is the best option.

2.3.4 Technical aspects

On a more technical level game engines and simulators don’t differ that much. Game engines are of course made for a specific game or type of games and hence are a bit less forgiving about what you can use it for. Simulators are constructed with versatility in mind can be used for a wide variety of purposes. A game is for the same reason equipped with all the parts needed for that game type such as AI, physics and network core. Simulators on the other hand are most often constructed as packages that can be interconnected in a more loosely manner and you only add the parts you need. As the creating company can’t supply parts for every specific need, only the most common packages are available. However, documented interfaces and a design plans for external components exits and in such the creation of specific modules can easily be conducted and then integrated into the simulation. In despite to games, simulators use standardized formats for objects. This increases the possibility to reuse objects and seamlessly move them from one simulation environment to another. The most used standard format for objects in simulations is the OpenFlight format.

Vega Prime as well as most other military targeted simulations can address very large areas, such as a world spanning well over 100x100 km. By hand creating such a large area is not something easily done and especially not with any larger detail. This is one of the main reasons why games don’t handle areas this large. In games where blending graphics is essential there is just no cost effective way of making an area that big detailed enough to support the demands of today’s players, hence there are none or few games supporting this. Exceptions for this are in the field of MMO’s (Massively Multiplayer Online) where the game areas found can be very large, but on the other hand details are then not the prime objective but rather player to player interaction.

2.3.5 Vega Prime

Vega Prime is a simulator created by MultiGen-Paradigm that mainly target military applications and usages. Vega Prime is a plane real-time 3D visualizer for which MultiGen-Paradigm as well as other companies has created add-ons that interoperate with Vega Prime. These add-ons can support Vega Prime with functionality such as AI, physics, network usage and HLA replications. There are also additional tools such as Creator Terrain Studio that help the designer create the simulated world. Together with these add-ons Vega Prime form a usable simulation with real world dynamics. If looking closer on Vega Prime there is a renderer or in this case a scene-graph which draws your world to the screen, Just as games Vega Prime use a technique called shaders (which are small programs doing per pixel changes directly on the graphics card) to create realistic looking lightning effects, such as reflection, refraction, heat, and effects such as the sun moving over the horizon and temporary blinding after
Background to Games and Simulations

looking straight into the sun. These small programs make a huge difference in the visual effect of the game or simulation.

Vega prime have a fully open source to the user and MultiGen-Paradigm or its resellers can support development teams with both experience and direct aid.

2.3.6 Unreal Engine

The unreal engine from EpicGames is one of the world’s most reused game engines and is available for licensing for any serious actor in the business. The engine exists in two main versions, 2 and 3 and also a subversion of the 2nd for game consoles. For the use of serious games Unreal Engine 2 is probably the best choice as it is a stable version since some year back. There is no ongoing development for the engine but as the full source code is supplied with the license, as well as access to their development network, adding other features to the engine is fully doable. Unreal engine 3 is still under ongoing development and in such is not stable and is more targeting future games for high-end pc and game consoles, such as Playstation 3 from Sony. Licensing any versions of the engine will grant access to their development network, a network both giving support in forms of forums and also a full manual and SDK (Software Development Kit) over the full engine. Parts of the development network can be accessed without a license as a means to support home development and moding for the Unreal Engine 2. (EpicGames, n.d.).

The Unreal Engine does not work that well with larger worlds. Worlds of 100x100km are not doable with the original code and have never been an intension by the developer. One game that uses such a large seamless world on the Unreal engine is Vanguard from Sigil Games Online. Sigil uses a modified version of the engine but after testing the game it is clear that there are zones at which loading take place so it is not fully seamless. The player is not presented with a loading screen when crossing a border but the game freezes for some seconds and the character is then dropped down on the other side of this invisible border.

The game engine is on the other hand very good with details so when intense graphics are needed it is a good option.

Connecting the Unreal Engine to other simulations and component is possible as there is a HLA interface created by MÅK but hints from a supplier of MÅK indicates that the solution is some what slow and is best used when replicating data out from the game.

The documentation for the Unreal Engines is extremely good and there are both examples and tutorials on almost all subjects. The engine has been in use for some years now and the documentation has grown to clear out all questions that a developer team can have. When licensing any of the engine versions support is supplied for as long as the licensee has ongoing development for a game and one year after shipping, other solutions can most likely be arranged. Support questions are directly answered by the development team.

EpicGames are one of very few actors in the game industry that have made supplying and creating a game engine their main business. It should be understood that Serious Games is not something that the game industry take any larger notice of as most Serious Games project don’t have any larger budgets and hence can’t afford the game engines. As such EpicGames primarily focus on game development and larger game companies (which on the other hand can be a company of only 60 people) they are probably not used to work with demands from companies such as Saab.
Background to Games and Simulations

This engine is very good in situations where a stable and mature game engine is requested, but it is not a simulation and it is not optimized for large spanning worlds. In that way it is probably not such a good tool to interconnect to other simulations as it can only work within its contained area/level. However most training problems and repetitive objectives can be conducted in a fun and controlled environment that don’t span over such large areas.
3 Problem description

Saab, as well as many other companies in the military industry, has come to look at games as a means for better looking visualizations for simulations and training tools for a cost that is less than today’s expensive simulators.

The need for this report and the work behind it have come from the problems that Saab, as well as others, have faced when trying to use games as parts in the development of military systems. The problem is that a game possibly has a goal that is not coherent with what the military intend to use it for. A game has a prime goal to entertain and the military, but also civil, systems have a goal to educate or by simulation evaluate a process of some sort.

To go from entertainment to education/testing, the game needs to be altered in some way. Integrating external components into game engines and in that way closing the gap between games and simulations is one way of making a more serious usage of games that are originally created for entertainment. It is unlikely that integrating external components into games will be the same thing as doing the same into well documented simulators or other types of systems that the military industry commonly work with. Even though games more and more have come do look like simulators in aspect to interior design and simulators are now using the same hardware and graphical enhancements as games, their prime goal are still very much different.

McGrath (2004) and McGrath (2005) showed that it is possible to integrate an external system into a game engine but they also pointed out numerous problems with this. Their implementation did not use an external component working in real-time and they did not work with the full source code. As the real-time usage can be seen as a likely request it is the next step to evaluate. If such an implementation shall stand any larger chance of success licensing the full source code is probably the only option. As McGrath (2004) and McGrath (2005) showed that the game engine can be quite dictating about design decisions.

The aim of this report and the work that follows is to evaluate the hardships in integrating external components into game engines.

The following limitations will be set for this work:

This report will target the usage of commercial game engines as they to a higher degree can be seen as one group with more of a common ground, compared to every other possible game engine created by hobbyists.

This implementation and the conclusions drawn from it correlate to implementation of real-time systems. It is likely far less problematic to integrate a non-real-time component.

3.1 Objectives

The following sub-goals or objectives have been setup for this assignment:

- Through the work at Saab get or create a realistic problem domain.
- Acquire a game engine with full source code.

---

3 Saab is a leading company in the military industry and a ‘real-world’ problem for Saab can be seen as a general enough problem.
Problem description

- Find a suitable external component that matches the problem requirements.
- Integrate the external component into the game engine in a successful way so that the real-time requirements are met.
- Evaluate the result and collect experiences throughout the assignment.

3.2 Targeted readers

The reader of this report is possibly somewhere in the beginning of an evaluation process of serious games, as a concept or on the hunt for a specific game engine but unsure of how to make a good selection and how to proceed. This report will focus on the possibilities of external integration but it might still prove to give valuable information and conclusion for other means.

3.3 Method

In order to successfully meet the sub-objectives and get the most realistic view of the problem and correct data, this research will follow the process of a full-blown integration project, for which the aims are closer described in chapter 4 and 5.

Chapter 4.1 will describe how the selection of a game engine can be done and at what level integrations such as this can take place.

The external component used must be representative for legacy systems in that it is not designed to be integrated with games. The selection of such a component is best decided by Saab as they know which components match the project mission the most.

As it is not the game engine or the external components own capabilities that are going to be evaluated the measure of success is put on a collective level where a local (program specific) bug or error is not as highly rated as a problem in industry-cultural differences.
4 Realization

4.1 Evaluation Criteria

Even if there are many ways of creating serious games there are four main approaches to conduct the process:

- Using a game as it is
- creating a mod using tools released with the game
- license the source code for a game engine
- building a game from scratch.

Depending on the design goal, and how much this overlap with existing games or game engines, appropriate decisions should be made. Creating a successful game is not something that just anyone is fit to do nor likely to have the budget for.

As with all development some sort of design plan is needed before any other work or decisions can be made. This design plan needs to explain and motivate every part of the game. At a first stage major things as; settings, usage of vehicles, details, real world accuracy, map size and other vital components need to be agreed upon. When there is a document that can explain the whole game and all parts that are in it, it is time to conclude what parts are vital for the essence of the game.

If giving the example of a rescue game one vital part might be the idea that two players can work together with one stretcher putting a patient on it and then carrying them away. Another part might be the possibility of a firefighter using a hose connected to a fire engine or hydrant. Such key parts that cannot be taken away without loosing the essence of the game must be clearly stated before moving on to the next phase.

Selecting the correct level of interaction is about knowing the product and knowing the market.

Figure 3 show the different levels of interaction that can be taken when engaging in the creation of serious games, starting from the simplest one.

![Fig. 3. Levels of interaction.](image-url)
4.1.1 Using the game as it is

This can be done when there exists a game that does exactly what is intended or if the game is used as part of something larger from where the serious aspect can be derived. If this level is acceptable this is without doubt the most cost effective stage as the initial cost will be around US $40 and there will be no additional development costs. However, the possibility to find such a game is not very big. Due to legal restrictions in using a game for commercial means there might be additional costs or fees depending on how it will be used.

4.1.2 Moding

Moding the game takes a little bit larger effort and ‘know how’ but is probably to a larger extent the right level to start from in most cases. Finding a game that fully matches a designer’s wish list is not very likely but finding a game that match the key essences of the design plan is crucial. At this level fundamental aspects of the game can’t be changed, for example: one can probably create a fire engine but if the original game has not been about firefighting the chance that a hose can be connected to it is very little as that was never a part of the original design. So once again knowing the market of games is important and without that knowledge there is no way of finding a game that matches. Initial cost for a serious game at this level is the same as in the first level, around US $40. Development costs are harder to put a price on as this very much have to do with the changes that need to be done and how much experience the development team have from game moding. If the serious game is taking place in the same game environment as the original game not much extra funding will be needed on new game models and graphics, so taking that into account when selecting a game can make things less costly. When the whole development and moding is complete and there is a final product there might be additional fees for using the game mod in commercial interests. Integrating external components at this level is possible and as this is the only affordable option for most companies there have been some work conducted at this level. Chapter 2.1.2 gave one example of such work.

4.1.3 Licensing a game engine

Licensing can be quite costly and there is much work left to be done before a new product emerge. Starting at this level is probably not the best idea unless very much is known both about the game engine and the serious game to be created. A firm knowledge about game development is also needed for any larger chance of success. Recommendations here are that the development team start off at the “Mod” level of the game and do as much as there can be done at that level and if they feel that this engine and all its tools are the right choice but it just lacks some features they can then move on and license the engine. One such feature could be the ability to replicate actions using HLA, or there might be a need for integrating external components or applications. Just as in the previous level the graphics are a big cost so if the original game graphics are usable that is a good thing. If working for the right reasons at this level one should understand that the development costs here are high. If the game engine need to be changed it is probably due to that it lacks some vital parts of your design, changing the original code in a good way takes both time and skill. Initial costs here are usually high and can range from US $350,000 to around US $600,000 for the more successful engines. There are also some with alot higher cost such as Warcraft III, which have a price tag at $3,750,000. One successful serious game that was developed at this level was America’s Army with a development cost of US $16 million. If the changes are large it might be a wise idea to put out the development on
a company that is used to work with the engine and have released previous successful titles on it. If integration of external components is vital then this might just be the right level to work with. It is likely that the component need to be integrated into more than one part of the game engine and unless the engine developer provide large parts with full source code to modders it is likely that this is the only way to go.

4.1.4 Starting from scratch

The last level is to completely abandon the reuse of games and create one from scratch which can be exceedingly expensive at least if the aim is to have it look similar to state of the art computer games. Creating this type of serious games from scratch is nothing that Saab or any other company that doesn’t have its main source of income in game development should try out. Initial cost at this level is $0 but development cost can rise to around US $40 million and that is with good and experienced game developers.

Working at this level might be good for smaller games that might be used in showrooms or at trade fairs. The creation of such small games can in a cost effective manner be carried out by students from schools with programs in game development such as University of Skövde.

With this in mind the first and last options can be ruled out for any larger project. Working with the two options left finding a game that closely matches the company/project’s needs is the next step to take. When one or more such games are found evaluating these games and their engines should be done through both the work of moding and also with respect to the possibility of feature licensing of the complete source code. Moding is a strait forward evaluation approach that doesn’t cost anything more than a single copy of the game and the time needed. If one feel that there is a need for modifications in the source code licensing might be the only way out. When evaluating a game engine for possible licensing there are some things that should be considered, even if the moding goes all fine.

Fig. 4. Levels of interaction with pricing.
4.2 Integration of external component

Integrating external components in game engines is an interesting way of enhancing the usages of legacy systems. As simulators and game engines come closer and closer to each other in a technical view using game engines for other means than entertainment is not far fetched.

4.2.1 Domain background

BF2 short for Battlefield 2 is one of the best selling games ever and is developed and produced by Digital Illusion CE AB (DICE), Sweden’s largest game developer.

Saab has done previous work with BF2 and has seen partial success doing so. Saab licensed the source code for BF2 well over a year before the game’s release date and as such the game and code was initially in a non-stable beta.

For this project Battlefield 2 have been selected as the game engine to be used. The decision was in this case based on availability as Saab already had licensed the full game engine. This license comes at a cost of SEK 4 million and is an opportunity not easily rejected.

The ‘real-world’ aim of this project in which integration of external components are going to be evaluated in is to have the dynamics of two or more boats in the game BF2 controlled by an external physics module called Simnon.

4.2.2 Battlefield 2 game engine

BF2 is a, what is generally referred to as a, First person shooter. The user plays the role as a soldier in a battle of some sort, ranging from about 16 to 64 simultaneous players divided on two different teams. The game is network based and the player sees everything from the soldier’s eye view or from an external camera view if using a vehicle.

The BF2 game engine is as most other game engines built up by modules that are interconnected to each other. The physics is one such module, which is then loosely connected to a dynamic object, such as a car or boat. From the way that this is designed one get the ability to create a new physics node that have some other features and connect this one instead of the default and get a new behavior for that object.

This feature is here used to enable the connection of our own physics node which in itself is connected, via a distribution interface, to a simulation program called Simnon.

4.2.3 Simnon a simulator of ship dynamics

Simnon is a stand-alone simulation program made by SSPA Sweden AB (Formerly know as Statens skeppsprovningsanstalt) that runs simulations of ship dynamics. This project has been working with a version that will fully simulate the behavior of a Stridsbåt 90. Stridsbåt 90 is a small Swedish naval transport and close combat vessel.

Simnon is written in Fortran a program language more suited for mathematical calculations than integrations with games but it has an interface for C/C++ function calls. Simnon is not originally constructed for the needs of this project and as such give a good example of the problems in integrating old legacy applications with game engines.
Simnon does not handle multiple simulations hence it is impossible to run more than one simulation in each program instance. To make things even more complicated it is not possible to go around this by starting more applications on the same computer. One can only run one instance of the application at the same time on a given computer. So what need be done to be able to have multiple simulations is to distribute the simulations onto other physical computers.

For this a distribution interface was created for BF2 and also a Simnon client program that enables Simnon to be run on another physical machine. The language used for this was C++.

Due to legal agreements with Dice no code from the server side may be exposed.

A more detailed explanation of this practical implementation and the code for the client side is described in chapter 5.

During the work with this implementation some things posed a distinctively larger threat to the project deadline than the rest of the problems. They might not be totally unique to game engine integration but they are factors that can be seen as general problems when working with game engine integration. Figure 5 show a representation of the extra work time that each problem posed. The column heights are fictive as the project time was not measured per hour and problem, but they give an indication and in what order they are ranked.

1. Legacy system not meant for games
2. Interface stability
3. Documentation
4. Support
5. Flexible components
6. Distribution of power

4.2.4 Legacy systems

As stated in 4.2.3 Simnon did not support multiple simulations and this was the sole reason for why this network-distributed interface was created. This was not a problem that came as a surprise but never the less it shows with great delicacy a problematic aspect of these kinds of integration. It does not matter how good the game engine is for external integration if the same cannot be said about the external component. The
uses of legacy systems with games pose another problem, which is over-all game speed. Not all military systems operate at 30 frames/s or higher and this can be a problem. The over all game speed is important for the gaming experience and if the game runs slower then 30 frames/s the experience get bad and the game is also much harder to control.

4.2.5 Interfaces do they last forever

In the work with BF2 and Simnon the first version used was a beta version and as work progressed at Dice new patches were released. However, these could not be implement as the interfaces changed and for every new patch most of the work had to be redone.

An interface that changes with every new patch is a nightmare for any external integration process.

The old beta-code was significant slower then the release-code which still hade not been used, and as this network-distribution did not make things better it was decided that the application should be updated to the most resent version of the code. For this to work most of Saab’s earlier work had to be rewritten to fit the new interfaces as well as class implementations.

The result was a much faster implementation but it came at a high cost in time, having a stable version to start with is likely a must or at least a great time/money saver.

4.2.6 Documentation

Dice as well as most other game companies do not spend much time writing comments or documentation. It is likely that there exist extensive documentation of gameplay and design in general but when it comes down to the code this is something rare in this business. A successful game engine is probably one of the most complex things ever coded, part from operating systems. Working with such a system with hundreds of files and many thousands of functions without any documentation what so ever is not time well spent. The lack of documentation and especially for new patches slowed down the development process, as there was no direct and easy way of knowing what had changed since the last patch. This was a problem that showed no mercy and had no real solution. During this integration process much time was spent on trail and error testing to understand how every system fit together. Dice employees answered some questions over mail but this could never replace the missing documentation.

4.2.7 Support

Many game engines are only supported through websites, and a large community of experienced hobbyists stands for the larger part of the support at hand. This can be just as good but you might have to expose more of the project to the masses if you want good help. In this case Saab is the only company to this date that have licensed the game engine for BF2 and as such the only help that can be given regarding the code is from Dice themselves. When the code was licensed no contract was signed regarding a support license and only an agreement was made about Saab buying support when it was needed.

This had been fine if the support was never needed or if Dice had not been such a successful game developer. Too much of the support control was now in the hands of Dice who were just about to release the successor of BF2, Battlefield 2142, and had little time for Saab’s requests.
Engaging a game project without a good support license might be more costly than paying for it at the start as fixing problems will take more time and hence every bug has the potential of being a larger long term cost. A support license/contract commits the game developer to support the game for the full length of your project, or at least for a known length of time.

If the engine was built around and for a specific game there is a good chance that the engine is only supported as long as there is development still going on against and for that game. If you don’t have a close relationship with your game partner you may well at some point find yourself very much alone. There are big differences in project cycles length if comparing games with military industry. A game can be forgotten after only a year or two.

When Saab delivers a system the support for it can normally range for up to 15-20 years, which is huge difference in compared to the life span and support of a game. Even if a game series live on it is not uncommon that the engine will be a new one or totally rewritten in a two to three year span, in order to keep up with hardware improvements and other games.

4.2.8 Flexible components

How well do the engine cooperate with the plans for network usage? Does it demand that one follow a certain path and don’t let you step outside? Is there any good way of changing the physical representations or adding your own? In this project a future goal was to integrate network usage in a way that one can connect all game engines and simulations to a large server, which handles and distribute the position of all game entities. In this case an engine taken in for consideration must have such a flexible network core that it either can be taken apart or rejected completely without this interfering with the rest of its components. BF2 have a well-designed physics implementation that was easy to change out. But it was instead not possible to have different physics for each object of the same type. The network code on the other hand is so intervened into the engine code that there is likely no way of changing that out with any greater success. When the game is created the code designers need to get this game to run fast as possible as this give more room for blending graphics. Getting things fast comes hand in hand with optimization, and optimizing will most likely come at the price of less flexibility.

4.2.9 The distribution of power

Every engine on a given hardware has a finite number of polygons that can be shown. Some can show more, some can show less but in the end the visual effect is more affected by the distribution of power. Where does the engine put its powers, is it in close up details or being able to show a larger landscape at once? Neither the ship nor the flight captain is likely interested in being able to see the burger menu on McDonalds but a big overview is demanded. To prove any use in such a situation the engine has to be able to sustain a world as large as 100x100 km or possibly even larger. In the case of a game where troops are securing the city core or a harbor the interior of McDonalds is more likely to be of interest than having the theoretical possibility of watching the landscape 20 km away. As most military systems work with huge areas using a game that can only handle a small area will prove to be somewhat crippling.

BF2 can work with maps that are approximately 3x3 to 5x5km depending on the number of objects and details on the map, in the work with the implementation larger
Realization

maps were requested so that it could be used for a more realistic costal scenario. This showed to be impractical as so large maps had never been a desire from Dice’s point of view and was not to any greater extent supported by the engine. Using a jet powered boat in an area of only 5x5km is not very meaning full.
4.3 Implementation

A case scenario of integrating an external physics module with a commercial game.

Chapter 4.2 gave a background and a less technical view of the integration work. Chapter 4.3 will instead give a closer and deeper explanation of the project.

4.3.1 A distributed interface

Three parts make up the distribution interface for the BF2 server. From the left as seen in figure 6 there is the SeamanPhysicsNode (SPN), SimnonInterface (SI) and the SimnonClientInterface (SCI).

![Diagram of distribution interface](image)

**Fig. 6. Layout of the distribution interface.**

The SeamanPhysicsNode is as the name may hint the physics node. The Physics node is a part of the original engine and it calculates and handles the physics for the connected object.

As BF2 and most games don’t do a full simulation of the world’s physics, cars have one type of physics and boats another. How the physics is calculated for each object is decided by its physics node. All dynamic objects (everything that can be interacted with or can move) have a physics node of some sort, through a common interface in

![Diagram of physics node](image)

**Fig. 7. Design of physics nodes.**

the Physics Node base class this can be any physics node that derive from the base class.
Realization

In this implementation a new physics node was created that use the same interface as the generic boat-physics node although the functions have a different implementation. The main difference between our physics node and the original is that ours don’t do any calculations.

4.3.2 SPN – SeamanPhysicsNode

The SeamanPhysicsNode work with both in and out going data, from and to Simnon. Out going data is extracted information from the game that Simnon need to make a correct simulation, such as water depth, throttle and steering angle etc. This information is in the SPN converted from BF2 structures into structures handled by Simnon and then passed onto the SI. In going data is the calculated information from Simnon, such as new position and orientation for the boat. This information is in the SPN converted from Simnon structure into data structures understood by BF2. The latest data is then saved in Physics Node interface -specific member variables and are later used by the game code to render the boat correctly.

4.3.3 SI – SimnonInterface

The SimnonInterface is a singleton network interface that takes care of sending and retrieving information between the physics node and the Simnon client interface. The information is sent using Winsock 2 TCP/IP with UDP packages, this works well as no packages are lost in this small closed environment. The interface only uses such small packages that they will never stand the risk of being split and the application is in it self also fault tolerant against packages lost or received in the wrong order. This tolerance comes from the fact that the physics is updated about 10 times more often than the screen image and as it is exact positions that are transmitted from Simnon the chance of seeing an erroneous position is 1/10 if there should be a bad package, which in itself is unlikely.

The SimnonInterface also keeps track of each and every connected Simnon client and dedicates physics nodes to boats as new Simnon clients connect and are available.

The potential late arrival (after the game session have started) of connected Simnon clients are not a problem as this have been taken care of in a way that the SI don’t connect the SPN to BF2 until the application have a free Simnon client connected. Until this happens BF2 take care of the physics with its own boat physics. Through this behavior the boats are usable at all time and as soon as a new Simnon client is connected it will be assigned a boat and the next time a driver enters that boat the SeamanPhysics node will be connected and the Simnon client will take care of the simulation instead of BF2. BF2 have a good physics module and it looks quite accurate even without the real simulation from Simnon, but the boat will behave as a propeller driven boat instead of a dual jet powered. Something mainly observed by the driver.
Realization

Here follow a conceptual workflow of how this connection process works.

When the SI is first initiated it also registers a windows callback function for the network so that the SI does not have to check for updates all the time. The callback is handled by the game loop and the SI is called and it processes the incoming data. The SI is listening for messages on port 9008.

Fig. 8. Connection workflow.

When the SI is first initiated it also registers a windows callback function for the network so that the SI does not have to check for updates all the time. The callback is handled by the game loop and the SI is called and it processes the incoming data. The SI is listening for messages on port 9008.
4.3.4 SCI – SimnonClientInterface

On the other side of the connection (on the client side) one find the SimnonClientInterface. The SimnonClientInterface is a stand-alone windows program that takes a single optional parameter, the IP address of the game server in the standard form of xxx.yyy.zzz.www. (Like: “SimnonClientInterface.exe 192.168.0.3”)

If nothing is entered the program will by default test if the server is located on the address of 192.168.0.1. Figure 9 show a simplified schematics of the data flow in the SCI. Up in the left we can see the WinMain, which is the main Windows program, from WinMain we connect to the game server and the SI. The SI will reply with a connection reply message and tell the client that it is connected to the server. From now on any correct message will be of the type Steering, Propulsion, Physics update or Reset. For each message there is a corresponding function in the SimnonCommunicator and the message is forwarded to the respective function. In the SimnonCommunicator the message is processed and then sent to Simnon.

Looking at this closer there are two larger parts that makes up the SimnonClientInterface. The main program WinMain and a smaller part called SimnonCommunicator. The main program is divided into several classes and most essential (apart from WinMain) are a network wrapper called SnailNet, a ServerProcess and also a file called netTypes.h. This is also seen in figure X.

To give a small introduction to these parts before looking closer on the code first out is SnailNet, which takes care of some of the more general network functions such as creating network addresses and initiating the Winsock.
Realization

The ServerProcess handles all incoming and outgoing information to and from the server side SI. And last there is the netTypes.h, which simply is a file where all globally used structures and constants are defined.

4.3.5 WinMain

Apart from the program loop and other functions that registers the program and work with the visual appearance, one of the more important functions in the WinMain is the programInit function, which with some cutting\(^4\) looks like seen below:

```c
void programInit( void )
{
    //Startup the log system.
    SnailNet::instance()->startLog("SimnonInterface.log");

    //Init the network and the socket.
    SnailNet::instance()->logString("starting system");
    SnailNet::instance()->initWinsock();
    SnailNet::instance()->initSocket(PORT_SERVERMAP_LISTENFOR_SERVER, socServer); //9019

    //Init the servers precess, most be done after initSockets to get valid socket numbers.
    ServerProcess::getSingleton()->setSocket(socServer);

    //Set the callback thread for the sockets for sending back a message on a "read" event.
    WSAAsyncSelect(socServer, hWnd, NET_READ_SERVER, FD_READ);

    //Start the simnon node.
    SimnonCommunicator::getSingleton()->init(".");
}
```

The port used to listen for the server is statically set to 9019 but this could easily be done dynamic by adding the port as an extra parameter in the program’s parameter list and having a default value of 9019 if nothing is entered. The program line could then look something like this: “SimnonClientInterface.exe 192.168.0.3 9001” which would start the client and let it know that the server can be found on the address 192.168.0.3 and that the server should reply to port 9001. This version lacks this feature for the only reason that there was no need for it at this point.

PORT_SERVERMAP_LISTENFOR_SERVER is a longer name for a constant

\(^4\) Code that is not essential for the implementation or its design, such code for the use of Singleton classes will not be presented in this report.
integer ("const int" in C/C++) with the value 9019 and socServer is a globally allocated Winsock 2 socket.

Two other important functions are the connection function and the shut down function. The connection function sends a message to the server side SI and let it know that a new Simnon client is ready for usage.

    ServerProcess::getSingleton()->sendConnectionMessage(g_serverIPAddress, 9008);

The shutdown function let the client user choose between a safe shut down and a fast shut down. If the safe shutdown option is used the SCI will sends a message to the server letting it know that this Simnon client will no longer be operational. The function takes a single parameter that is the time before shut down in ms.

    ServerProcess::getSingleton()->sendShutdownMessage(0);

The fast shut down on the other hand will simply shut down the client program as fast as possible without letting the server know that this client is no longer responding. In this version the fast shut down is not recommended as this will lead to that the boat that had this Simnon client dedicated to it will be useless for the rest of the game session. The boat will simply not get any new updates and will there for no longer move, it is still usable in any other way and the program will not crash but there is no way of getting that boat functional again.

If the safe shutdown is used the server side SI will deallocate the SCI from the boat and reconnect BF2’s original physics node, and if there is a free Simnon client connected or if one connects the next time the driver enters the boat it will once again be under the control of a Simnon client.

When a new network message arrive to the SCI the windows callback function will notify the SCI by sending it a program message, which is being handled, in the program main loop and the ServerProcess is activated to read the actual network message.

    ServerProcess::getSingleton()->handleIncMessage();

### 4.3.6 ServerProcess

The ServerProcess handles all incoming traffic from the server side SI. The following functions is of interest in the ServerProcess:

    public:
    
    // Set the socket for the process
    int setSocket(IN SOCKET &socSocket);
    
    // Main process that translate networkmessages and then handle them.
    void handleIncMessage(void);
    
    // Send back the latest data from the simnon simulation
    int sendSimnonData(IN Vector3D p, IN Vector3D o, IN Vector3D v);
    
    // Send a message to the server letting it know that this Simnon client is up.
    int sendConnectionMessage(IN char FAR * pIPAddress, IN int port);
    
    // Send shutdown message to server
    int sendShutdownMessage(IN int InTimeMS);
    
    private:
    
    static ServerProcess* m_pServerProcess;
    struct sockaddr_in m_serverAddr;
    int m_serverLen;
    SOCKET m_socSocket;


Realization

Starting from the first function the implementation looks like this:

```c
int ServerProcess::setSocket( SOCKET &socSocket )
{
    m_socSocket = socSocket;
    return 0;
}
```

The member variable m_socSocket is set as the reference socSocket and the function then returns 0.

The next function named “handleIncMessage” is the largest and also the core of this class. Implementation goes as follows:

```c
void ServerProcess::handleIncMessage(void)
{
    char buf[sizeof( NET_MSG_MAXSIZE )];
    m_serverLen = sizeof( m_serverAddr );
    NET_MSG_PHYSICSUPDATE* pSimnonPhysicsMessage;
    NET_MSG_COMMAND_DATA* pPropulsionMessage;
    NET_MSG_COMMAND_DATA* pSteeringMessage;
    NET_MSG_RESET_DATA* pSimnonResetMessage;
    NET_MSG_DATASIZE* pDataMessage;

    // Read the incoming data from the connected socket.
    if(recvfrom(m_socSocket, buf, sizeof(NET_MSG_MAXSIZE), 0, (struct sockaddr*)&m_serverAddr, &m_serverLen))
    {
        pDataMessage = (NET_MSG_DATASIZE*)buf;
        switch( pDataMessage->type )
        {
            //client connection request
            case NET_DATATYPE_CONNECTION_REPLY:
                break;
            case NET_DATATYPE_RESET:
                pSimnonResetMessage = (NET_MSG_RESET_DATA*)buf;
                SimnonCommunicator::getSingleton()->reset(pSimnonResetMessage);
                break;
            case NET_DATATYPE_COMMAND_PROPULSION:
                pPropulsionMessage = (NET_MSG_COMMAND_DATA*)buf;
                SimnonCommunicator::getSingleton()->propulsionCommand(pPropulsionMessage);
                break;
            case NET_DATATYPE_COMMAND_STEERING:
                pSteeringMessage = (NET_MSG_COMMAND_DATA*)buf;
                SimnonCommunicator::getSingleton()->steeringCommand(pSteeringMessage);
                break;
            case NET_DATATYPE_PHYSICSUPDATE:
                pSimnonPhysicsMessage = (NET_MSG_PHYSICSUPDATE*)buf;
                SimnonCommunicator::getSingleton()->updatePhysics(pSimnonPhysicsMessage);
                break;
            default:
                break;
        }
    }
}
```

In the implementation the message is first collected and then cast into a generic message of the type NET_MSG_DATASIZE the new casted message is saved in the variable pDataMessage. The reason for this is that it is not yet know of what type the incoming message is so it is impossible to directly cast it correctly. To handle this all messages start out with an integer that identifies what type the message is so that it
after a check can be cast correctly. What this initial cast does is cast it to a message type that only contains this first integer that tells us the type and all other information is truncated away. The type is then checked in a case statement where all possible types are present. For each message type the incoming message is casted to its correct form and handled in an appropriate way, in this case sent to the SimnonCommunicator.

When Simnon have calculated new values for the boat the information need to be sent back to the game server, this is the job of the sendSimnonData function. The function takes three vector parameters, which are the position, orientation and velocity of the boat. The function looks as follows:

```c
int ServerProcess::sendSimnonData(Vector3D p, Vector3D o, Vector3D v)
{
    NET_MSG_RESET_DATA* pUpdateMessage = new NET_MSG_RESET_DATA;
    pUpdateMessage->type = NET_DATATYPE_SETSIMNONDATA;
    pUpdateMessage->position = p;
    pUpdateMessage->orientation = o;
    pUpdateMessage->velocity = v;

    if(sendto(m_socSocket, (char*)pUpdateMessage, sizeof(NET_MSG_RESET_DATA), 0, (struct sockaddr*) &m_serverAddr, m_serverLen) == SOCKET_ERROR)
    {
        delete pUpdateMessage;
        return -1;
    }
    delete pUpdateMessage;
    return 0;
}
```

The three vectors are copied over to a network structure and that structure is then sent to the server side SI.

The next function send a connection message to the server letting it know that this Simnon client is ready for work, the implementation looks as follows:

```c
int ServerProcess::sendConnectionMessage(IN char FAR * pIPAddress, IN int port)
{
    // Create the Internet address from the IP number
    SnailNet::instance()->initAddress(port, pIPAddress, m_serverAddr);
    m_serverLen = sizeof(m_serverAddr);
    NET_MSG_CLIENT_CONNECTION* pConnectionMessage = new NET_MSG_CLIENT_CONNECTION;
    pConnectionMessage->type = NET_DATATYPE_CONNECTION;
    strcpy(pConnectionMessage->username, "user");
    strcpy(pConnectionMessage->password, "pass");
    pConnectionMessage->iClientPort = PORT_SERVERMAP_LISTENFOR_SERVER;

    if( sendto( m_socSocket, (char*)pConnectionMessage, sizeof(NET_MSG_CLIENT_CONNECTION), 0, (struct sockaddr* )&m_serverAddr, m_serverLen ) == SOCKET_ERROR )
    {
        delete pConnectionMessage;
        return -1;
    }
    delete pConnectionMessage;
    return 0;
}
```

In the first lines a valid Internet address-tag is created from the ip-address and port, the information is then saved in the class’s member variable “m_serverAddr”. After this a connection message is created of the type NET_MSG_CLIENT_CONNECTION. The structure is filled with the necessary information and then structure is sent to the server side SI (If the given connection information is correct).
The last function in this class is the shutdown function, which informs the server side SI that the Simnon client is about to shut down. The function takes a single parameter which is the time until shutdown in ms. As the SI have no way of knowing the exact time the message was created and sent the ms-time should not be taken as a solid truth by the SI or game server.

```cpp
Int ServerProcess::sendShutdownMessage(IN int iInTimeMS)
{
    NET_MSG_SHUTDOWN* pShutdownMessage = new NET_MSG_SHUTDOWN;
    pShutdownMessage->type = NET_DATATYPE_DISCONNECTION;
    pShutdownMessage->time = iInTimeMS;
    if( sendto( m_socSocket, ( char* )pShutdownMessage, sizeof( NET_MSG_SHUTDOWN ), 0, ( struct sockaddr* ) &m_serverAddr, m_serverLen ) == SOCKET_ERROR )
    {
        delete pShutdownMessage;
        return -1;
    }
    delete pShutdownMessage;
    return 0;
}
```

Once again a network structure is created and filled with information and then sent to the server side SI. All structures that are used for sending information over the network are identified by their names and that the name is also in uppercase. Such as NET_MSG_SHUTDOWN and NET_MSG_CLIENT_CONNECTION. NET_MSG let us know that this is a network message structure and that the structure can be found in the netTypes.h file.

### 4.3.7 netTypes.h

The last part that can be seen as a part of the WinMain is the netTypes.h, netTypes.h is a file that is known to most classes as it defines many of the variables and structures that are used globally. As netTypes.h does not contain any functions the full code is not shown here but can instead be seen in Appendix 1.

The information declared in this file is also strongly used by the network wrapper SnailNet.

### 4.3.8 SnailNet

To be able to send any information through the means of the network the network need to be initiated and the sockets used need to be activated and functional with the OS. SnailNet takes care of the things that are generic to most computer traffic over the Internet, at least for UDP packages.

The implementation for this class is rather long and not essential for this SCI in any other way then that it works, and some of the functions are not used at this point so the implementation of this part is placed in the appendix. For a closer look at the initialization of Winsock and the use of sockets Mulholland and Andrew (2002) give a good walk-through that can aid the understanding of the code.

### 4.3.9 SimnonCommunicator

The other somewhat smaller but more important part is the SimnonCommunicator. The SimnonCommunicator can be seen as another part of the physics node and it
Realization

takes care of communication with Simnon through its dlls and interface. In the SimnonCommunicator the following functions are of interest:

```c++
//convert a SCI Vector3D struct to a Simnon XYZ struct.
void convert3DToXYZ(const Vector3D v1, VectorXYZ &v2);

//convert a Simnon XYZ struct to a SCI struct of type Vector3D.
void convertXYZTo3D(const VectorXYZ v1, Vector3D &v2);

//initiate Simnon and tell it in which directory it can find its settings files.
void init(std::string workDir);

//reset Simnon between start and stop.
void reset(NET_MSG_RESET_DATA* pMessage);

//give Simnon a propulsion command.
void propulsionCommand(NET_MSG_COMMAND_DATA* pMessage);

//give Simnon a steering command.
void steeringCommand(NET_MSG_COMMAND_DATA* pMessage);

//update the pysics and get a new orientation and location for the boat.
void updatePhysics(NET_MSG_PHYSICSUPDATE* pMessage);
```

SimnonCommunicator feeds Simnon with what information it has and Simnon makes its calculations and return a new position and orientation for the boat. This new information is then transferred back through the main part of the program to the SimnonInterface on the game server and the boat is later updated for all to see. Implementation wise SimnonCommunicator looks like this:

```c++
void SimnonCommunicator::convert3DToXYZ(const Vector3D v1, VectorXYZ &v2)
{
    v2.X = v1.x;
    v2.Y = v1.y;
    v2.Z = v1.z;
}

void SimnonCommunicator::convertXYZTo3D(const VectorXYZ v1, Vector3D &v2)
{
    v2.x = v1.X;
    v2.y = v1.Y;
    v2.z = v1.Z;
}

void SimnonCommunicator::init(std::string workDir)
{
    ::SimnonStart(workDir.c_str());
    ::SimnonSimulateInit();
}

void SimnonCommunicator::reset(NET_MSG_RESET_DATA* pMessage)
{
    m_timer->start();
    VectorXYZ p;
    VectorXYZ o;
    VectorXYZ v;
    SimnonCommunicator::convert3DToXYZ(pMessage->position, p);
    SimnonCommunicator::convert3DToXYZ(pMessage->orientation, o);
    SimnonCommunicator::convert3DToXYZ(pMessage->velocity, v);
    ::SetInitialLocation( p );
    ::SetInitialOrientation( o );
    ::SetInitialBodyVelocity( v );
    ::SetCurrentVelocity( 0.0 );
}

void SimnonCommunicator::propulsionCommand(NET_MSG_COMMAND_DATA* pMessage)
{
    
    
    
```
Realization

```cpp
::SetPropulsionCommand(pMessage->command.k, pMessage->command.s);
}
void SimnonCommunicator::steeringCommand(NET_MSG_COMMAND_DATA* pMessage)
{
    ::SetSteeringCommand(pMessage->command.k, pMessage->command.s);
}
void SimnonCommunicator::updatePhysics(NET_MSG_PHYSICSUPDATE* pMessage)
{
    m_timer->stop();
    double time = m_timer->getDeltaTimeInSec();
    if(time > 1)
        time = 1.000;
    if(time < 0.000001)。
        time = 0.000001;
    m_timer->start();
    ::SimnonSimulateNextStep(time);
    if ( pMessage->physicsUpdate.waterDepth >= 0.0f )
        {
            ::SetWaterDepth( pMessage->physicsUpdate.waterDepth );
        }
    else
        {
            ::SetWaterDepth( 0.0 );
            ::SetPropulsionCommand( 1, 0.0 );
            ::SetPropulsionCommand( 2, 0.0 );
            ::SetCurrentVelocity( 0.0 );
        }[n]
    VectorXYZ p = ::GetLocation();
    VectorXYZ o = ::GetOrientation();
    VectorXYZ v = ::GetBodyVelocity();
    Vector3D P;
    Vector3D O;
    Vector3D V;
    SimnonCommunicator::convertXYZTo3D(p, P);
    SimnonCommunicator::convertXYZTo3D(o, O);
    SimnonCommunicator::convertXYZTo3D(v, V);
    double testX = P.x;
    double testY = P.y;
    double testZ = P.z;
    ServerProcess::getSingleton()->sendSimnonData(P, O, V);
}
```

Function calls that start with “::” are calls to the Simnon interface, as Simnon was not created for this assignment the interface is larger than what is used here but as that was never used it is not presented in this report.

As probably noticed the applications don’t distribute the new information to the game clients, only to the game server and then the BF2 engine is left to take care of its own clients. This is a very stable and easy approach as the distribution-interface doesn’t have to care about the game clients at all.
5 Conclusions

5.1 Summary

Games can in many ways be used by the military industry but good planning and preparations are essential for a success. The use of games and game engines should not be seen as a universal tool that will result in a good solution in every case. There are lots of areas where games truly can make a good transition into more serious areas but advanced simulations with multiple integrated parts are to this date not one of them. Games lack the interoperability design thinking and are totally inferior to advanced simulations when it comes to reuse of models and loading real world data into the simulated world. The use of automated world generating processes in simulations is to a large part what makes them look visually inferior to games, as the graphics in games are hand made whereas large simulations use real map data and flight photos. So using a game engine will not by itself result in a better-looking application. Using a game engine is most often only the better option if what is created is also a game of some sort and the fun factor is more important than truth to world constants.

The best economical solution is the one where the planed solution to the largest degree corresponds with the original game design. For that reason it is very important to plan well and to find out what elements of the planed solution is most important, and then try to find the game that to the largest degree resembles that design.

There are four ways of making use of a game or game engine, firs there is the possibility of finding a game that does exactly what it is needed for. Second one can “mod” the game, using the game as it is distributed but changing the game world with new contents such as graphics, objects, and environment. It is possible to integrate external components at this level as McGrath (2004) and McGrath (2005) showed. Third option is to license the game code, which cost a lot of money but give you total freedom, but a very complex freedom. Working at this level is probably the only option if more advanced integrations are required. Last is the option of creating a game from scratch, which for large projects is too expensive but can be a good option for small games for PR and trade fairs.

When a game with a similar design have been found that game or game engine should be evaluated, and the same goes for the external component. Some key elements that need to be evaluated for have been presented as:

Distribution of power

1) Legacy system not meant for games; not all applications or components work in a way that can efficiently be integrated into game engines.

2) Interface stability; how stable are the interfaces. When interfaces change things need to be rewritten.

3) Documentation; having a solid documentations to the code and engine is nothing that should be taken for granted as most game developers don’t spend much time writing comments or any other form of documentation if not necessary.

4) Support; without a solid support license every single bug have the potential of taking down the project. Also support time in the military industry can span up to 15-20 years, a game can be forgotten after only 2.
Conclusions

5) Flexible components; engine-components such as network and physics can be hard to disconnect and replace if the game has a high internal coupling. If the engine is going to be integrated with external systems or components, taking the time to find a fitting engine with low internal coupling will pay back almost immediately.

6) Distribution of power; some games work with details others with large areas and some are in between. In this case the external component was for boat dynamics but the game cannot support map large enough for this to be of any greater use.

As using games is not always the best choice the easiest way of checking whether a game or a simulation is most at hand is looking closely to find out which one most resembles what is requested. Is the game closer than the simulation, go with it but don’t just look at the graphics that is just a tiny bit of the larger piece. There is also the network, teams, GUI’s and everything else that might have to be changed to take in mind.

When using a game engine, one will also face the problems of reusability. The most used standard format for objects in simulations is the OpenFlight format, if using this, objects can move seamlessly from one simulation environment to the next. In games there is no such global standard and for every game engine the objects need to be reported. In a large company like Saab reusability is a factor as more than one division is creating objects and working with simulations and being able to easily implement new units is essential for the versatility of the product.

Simnon was successfully integrated into BF2 and multiple boats were connected and controlled by Simnon. As Simnon cannot work with multiple simulations in one instance a distribution interface was created and the Simnon clients were distributed onto other physical computers. Information about the BF2 players position was also extracted and fed to a real command and control system where the same real life area was showed as the player map represented and you could on the command and control system track the units movements. This last feature was not create for this integration test and is therefore not presented in this report.

5.2 Discussion

Giving Vega Prime a quick evaluation from the key issues that was found we find that Vega Prime with ease clear all these points as it is targeted for military use. There are stable interfaces as the code is highly modulated and the design is created with modifications as a prime factor. Support issues are not a factor as most of their licensees have projects spanning over tens of years. Vega Prime is not the best engine for close up details but instead work very well with large worlds over 300x300 km. Connecting Vega Prime to other simulations and components is not a problem since it, with the integrated VR-Link (created by MÅK), can distribute information using DIS/HLA. The documentation for Vega Prime is good, they have many other clients that can possibly run into the most common problems, and hence there might already be a solution to them.

If what really is sought after is a simulation with a high level of visual realism and not really an entertaining game then Vega Prime is a good option. In its original form Vega Prime will not offer all the things that are given in a game where everything is complete and ready to use, but what you do get is better control regarding design issues.
Giving the Unreal Engine the same quick evaluation as Vega Prime one finds that the Unreal Engine also clears most of these. At least for version 2 of the engine there are stable interfaces as there is no longer any new development on the engine. Support is not a problem as EpicGames still give support for their old engines as well as their new ones. McGrath (2004) and McGrath (2005) used the Unreal Engine for their UnrealTraige project and came to the conclusion that games still lack the interoperability thinking to be used as a great simulation tools. They also argued that it is hard work getting a game engine to work in any other way then its original design. They did on the other hand not work with the full source code as this project did and in such did not have the same options for changes. Even so there is truth to their words as game engines are so complex that any changes done to the original design will most likely only come to look as bad hacks.

The validity of this report should be seen with the eyes of the military industry which for the time being have a some what different view of the subject of serious games, compared to that of others within the field. Most other and especially those involved within the field of GBL want to harness the good learning factor that most games have. In such they don't always want to use games for the sake of being games but rather that the games industry have found a very good way of getting the users attention. They don't strive to use games as a product but the element that makes games so engaging and fun. The military industry and also most other non-GBL oriented parties try to use games as parts of simulations as cost-effective alternatives to the normally used simulators and trainers. It is likely that this view will change as more tests are undertaken on the subject and hopefully the military industry will change focus and to a larger extent use games for what they are and find situation where that work really well.

Though this work showed both good potentials for serious games with external components as well as obstacles that need further examination this work have only been validating the needs at one company. Saab is one of the largest companies in the military industry when it comes to technology and software development and in so represent the business well but not in total. The test case that was carried out at Saab generated good problems to test and evaluate but it was only one game engine and one external component. Larger tests could possibly show better surety to repeatedly occurring problems.

### 5.3 Future work

This implementation show that it is not problem free to integrate legacy systems into games. Further work need to be conducted to setup requirements that need to be met by legacy systems to have them efficiently integrated into games, as well as simulators in general.

In the military industry there exists standards for most about everything. In the game business it is the direct opposite, further examinations need to be done about how game engines can use the same standards as OpenFlight for models and DIS/HLA for replicating entity information between systems.

This report only looks on the subject of serious games from the developers perspective and no research is done about how a possible customer or client in the military business will embrace a product created on a game, or with a game like design. It is fully possible that a great product will be neglected due to lack of marketing of the concept of serious games. This type of work therefore needs to be
Conclusions

complemented with research on the awareness of the subject and the attitude on customer side parties.

More rigid evaluations of specific game engines are needed and how well they may work for integration processes and serious game development. The conclusions and advices from this report could be used, as a foundation for evaluation but also other requests should be met.

Make a more general list of demands on a game engine for serious games, this report take in the need from Saab and though it may be a general problem it does not include all aspects of possible problems in a larger problem domain. Though knowing what is wrong is a good start it is not very useful with out a roadmap with suggestions to how to fix it.
Conclusions

References


Conclusions

MÄK., 2004 MÄK Technologies integrates HLA networking interoperability in epic games’ Unreal Engine.

Unreal Engine 2., developer network. [online]
Available from: http://udn.epicgames.com/Two/WebHome/
[cited 26 Dec 2006]


Appendix 1

SnailNet.h

// Returns a pointer for the Singleton.
static SnailNet* instance(void);  // Don't forget to start the log if you want to use it.

// Start the log tool.
void startLog(IN char FAR * cLogFileName);

// Give the log a string to log, the log must be started to use this.
void logString(IN char FAR * string, ...);

// Initiate the winsock (2.0).
SN_RESULT initWinsock(void);

// Init a socket that listen for data communication on a specific port. use for retrieving data from network.
static SN_RESULT initSocket(IN int iPort, OUT SOCKET &socSocket);

// Init a socket that you can only use to send data with.
SN_RESULT initSocket(OUT SOCKET &socSocket);

// Init an inet address, used for sending data (like an address on a letter). Takes an address in the form that you receive from other messages, so it is good for replies.
SN_RESULT initAddress(IN int iPort, IN in_addr address, OUT struct sockaddr_in &inetServAddr);

// Init a inet address, used for sending data (like an adress on a letter). Takes an address in the form of an IP address, good for starting communication.
SN_RESULT initAddress(IN int iPort, IN char FAR * pIPAddress, OUT struct sockaddr_in &inetServAddr);

// Creates a 32 chars long key that can be used for verifying a user or communication. Right now it just creates a static string that matches what the "checkKey" is looking for, so not finished.
SN_RESULT createKey(OUT char cKey[32]);

// Used for verifying that a key is a in fact created with "createKey" and that the user is legitime. Right now it just checks for a statick string, so not finished.
SN_RESULT checkKey(IN char cKey[32]);

// Checks a username and password against a database. Does not do that at the moment, only correct name and passwords that can be used are "user" and "pass".
SN_RESULT checkUserNamePassword(IN char cUserName[16], IN char cPassword[16]);

// Compares two addresses to see if they are the same, like if a sender is the same as last time or an address you have saved.
bool compare(sockaddr_in addr1, sockaddr_in addr2);

SnailNet.cpp

// Partly from Mulholland, Andrew., 2002. Developer's guide to multiplayer games
SnailNet* SnailNet::instance(void)
{
    static SnailNet instance;
    return &instance;
}

void SnailNet::startLog(IN char FAR * cLogFileName)
{
    m_cLogfileName = cLogFileName;
    FILE* logFile;
    time_t current = time(NULL);
    if( ( logFile = fopen(m_cLogfileName, "w")) != NULL)
    {
        fprintf(logFile, "Log file started %s", ctime(&current));
        fclose(logFile);
    }
Conclusions

void SnailNet::logString(IN char FAR * string, IN ...)
{
    char buf[1024];
    va_list ap; // va_list is a redefined char*
    va_start(ap, string);
    vsprintf(buf, string, ap);
    va_end(ap);

    FILE* logFile;

    time_t current = time(NULL);
    if((logFile = fopen(m_cLogfileName, "a")) != NULL)[
        fprintf(logFile, "-> %s: %s", ctime(&current), buf);
        fprintf(logFile, "\n");
        fclose(logFile);
    ]
}

SN_RESULT SnailNet::initWinsock(void)
{
    WORD versionRequested;
    WSADATA wsaData;
    DWORD bufferSize = 0;
    LPWSAPROTOCOL_INFO SelectedProtocol;
    int NumProtocols;

    // Start WinSock2. If it fails, we do not need to call WSACleanup().
    versionRequested = MAKEWORD(2, 0);
    int error = WSAStartup(versionRequested, &wsaData);
    if(error)
    {
        return SN_FAIL;
    } else
    {
        // Confirm that the WinSock2 DLL supports the exact version
        // we want. If not, call WSACleanup().
        if(LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 0)
        {
            WSACleanup();
            return SN_FAIL;
        }
    }

    NumProtocols = WSAEnumProtocols(NULL, NULL, &bufferSize);
    if((NumProtocols != SOCKET_ERROR) && (WSAGetLastError() != WSAENOBUFS))
    {
        WSACleanup();
        return SN_FAIL;
    }

    // Allocate a buffer, call WSAEnumProtocols to get an array of
    // WSAPROTOCOL_INFO structures.
    SelectedProtocol = (LPWSAPROTOCOL_INFO) malloc(bufferSize);
    if (SelectedProtocol == NULL)
    {
        WSACleanup();
        return SN_FAIL;
    }

    // Allocate memory for protocol list and define what protocols to look for.
    int *protos = (int *) calloc(2, sizeof(int));
Conclusions

protos[0] = IPPROTO_TCP;
protos[1] = IPPROTO_UDP;

NumProtocols = WSAEnumProtocols(protos, SelectedProtocol, &bufferSize);
free(protos);
protos = NULL;

free(SelectedProtocol);
SelectedProtocol = NULL;

if(NumProtocols == SOCKET_ERROR)
{
    WSACleanup();
    return SN_FAIL;
}

return SN_OK;

SN_RESULT SnailNet::initSocket(IN int iPort, OUT SOCKET &socSocket)
{
    struct sockaddr *servAddr;
    struct sockaddr_in *inetServAddr;

    // Create the socket
    socSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if(socSocket < 0)
    {
        return SN_FAIL;
    }

    // Allocate memory for the address structure and set it to zero.
    servAddr = (struct sockaddr *) malloc(sizeof(sockaddr));
    memset((char *) servAddr, 0, sizeof(sockaddr));

    // Fill the address structure.
    servAddr->sa_family = (u_short) AF_INET;
    inetServAddr = (struct sockaddr_in *) servAddr;
    inetServAddr->sin_port = htons((u_short) iPort);

    // Bind the address information to the socket.
    if(bind(socSocket, servAddr, sizeof(sockaddr)) == SOCKET_ERROR)
    {
        free(servAddr);
        return SN_FAIL;
    }

    free(servAddr);
    servAddr = NULL;

    return SN_OK;
}

SN_RESULT SnailNet::initSocket(OUT SOCKET &socSocket)
{
    socSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if(socSocket < 0)
    {
        return SN_FAIL;
    }

    return SN_OK;
}

int SnailNet::initAddress(IN int iPort, IN in_addr address, OUT struct sockaddr_in inetServAddr)
{
Conclusions

```
memset((char *) &inetServAddr, 0, sizeof(inetServAddr));
inetServAddr.sin_family = AF_INET;
inetServAddr.sin_port = htons((u_short) iPort);
inetServAddr.sin_addr = address;
return SN_OK;
}

SN_RESULT SnailNet::initAddress(IN int iPort, IN char FAR * pIPAddress, OUT struct
sockaddr_in &inetServAddr)
{
    u_long inetAddr = inet_addr(pIPAddress);
    memset((char *) &inetServAddr, 0, sizeof(inetServAddr));
inetServAddr.sin_family = AF_INET;
inetServAddr.sin_port = htons((u_short) iPort);
inetServAddr.sin_addr.S_un.S_addr = inetAddr;
return SN_OK;
}

SN_RESULT SnailNet::createKey(OUT char cKey[32])
{
    strncpy(cKey, "J0o6Igja3gREa91aWWnw3kv8ejiaeekl", 32);
    return SN_OK;
}

SN_RESULT SnailNet::checkKey(IN char cKey[32])
{
    if(strcmp(cKey, "J0o6Igja3gREa91aWWnw3kv8ejiaeekl") == 0)
    {
        return SN_OK;
    }
    else return SN_FAIL;
}

SN_RESULT SnailNet::checkUserNamePassword(IN char cUserName[16], IN char
Password[16])
{
    if(strcmp(cUserName, "user") == 0 && strcmp(cPassword, "pass"))
    {
        return SN_OK;
    }
    else return SN_FAIL;
}

bool SnailNet::compare(sockaddr_in addr1, sockaddr_in addr2)
{
        return true;
    else
        return false;
}
```

SnailNet::~SnailNet(void){}

netTypes.h:

typedef int HOSTID;
typedef int GENERIC_MSG;
typedef int SN_RESULT;
const SN_RESULT SN_OK = 0;
const SN_RESULT SN_FAIL = -1;
const SN_RESULT SN_BAD_INPUT = -2;
const u_int NET_READ_SERVERMAP = WM_USER;
const u_int NET_READ_SERVER = WM_USER + 1;
const u_int NET_READ_CLIENT = WM_USER + 2;
```
const u_int NET_READ_SOCKET = WM_USER + 3;
const u_int MYWM_NOTIFYICON = WM_USER + 4;
const int PORT_SERVERMAP_LISTENFOR_SERVER = 9019;

//------------------------------------------------------------
struct Vector3D
{
    double x;
    double y;
    double z;
};
//------------------------------------------------------------
struct ResetData
{
    Vector3D position;
    Vector3D orientation;
    Vector3D velocity;
};
//------------------------------------------------------------
struct Command
{
    int k;
    double s;
};
//------------------------------------------------------------
struct PhysicsUpdate
{
    float deltaTime;
    float waterDepth;
};
//------------------------------------------------------------
typedef struct
{
    int type;
    char fill[512];
} NET_MSG_MAXSIZE;
//------------------------------------------------------------
typedef struct
{
    int type;
    HOSTID toId;
    HOSTID fromId;
    int size;
} NET_MSG_DATASIZE;
//------------------------------------------------------------
typedef struct
{
    int type;
    int time;
} NET_MSG_SHUTDOWN;
//------------------------------------------------------------
typedef struct
{
    int type;
    char username[16];
    char password[16];
    int iClientPort;
} NET_MSG_CLIENT_CONNECTION;
typedef struct
{
    int type;
    char clientKey[32];
    int status;
} NET_MSG_CONNECTION_REPLY;

typedef struct
{
    int type;
    Vector3D position;
    Vector3D orientation;
    Vector3D velocity;
} NET_MSG_RESET_DATA;

typedef struct
{
    int type;
    Command command;
} NET_MSG_COMMAND_DATA;

typedef struct
{
    int type;
    PhysicsUpdate physicsUpdate;
} NET_MSG_PHYSICSUPDATE;

#define NET_DATATYPE_CONNECTION     1
#define NET_DATATYPE_DISCONNECTION    2
#define NET_DATATYPE_CONNECTION_REPLY    3
#define NET_DATATYPE_GETSIMNONDATA    10
#define NET_DATATYPE_SETSIMNONDATA    11
#define NET_DATATYPE_RESET      12
#define NET_DATATYPE_COMMAND_PROPULSION   13
#define NET_DATATYPE_COMMAND_STEERING    14
#define NET_DATATYPE_PHYSICSUPDATE    15
#define NET_DATATYPE_DATASIZE     100
#define NET_DATATYPE_GENERIC     200
#define NET_CONST_CONNECTION_ACCEPTED    900
#define NET_CONST_CONNECTION_ALLREADYCONNECTED  901
#define NET_CONST_DISCONNECTED     902
#define NET_CONST_DISCONNECTED_FROM_REQUEST   903