

## Index

Games Definitions.....	3
Massively Multiplayer Online Game (MMOG).....	3
Massively Multiplayer Online Role-playing Game.....	4
Academic attention.....	4
Real-time Strategy (RTS).....	4
Typical Game Components.....	5
Game Design.....	6
Design Patterns.....	6
Initial Documentation.....	7
Technical Decisions.....	8
Networking.....	8
Communication Architectures.....	9
Problem of Server-Network: Scalability.....	10
Security.....	10
Packet and Traffic Tampering.....	11
Information Exposure.....	11
Design Defects.....	11
Artificial Intelligence.....	12
Classical techniques and topics.....	12
Genetic Algorithms.....	12
General Design Questions.....	14
Connection Issues.....	14
Operating Platform Issues.....	14
Technologies issues.....	15
Business Issues.....	15
Project.....	17
MMOG Idea.....	17
Goals.....	18
Networking.....	18
Artificial Intelligence.....	18
System Logic.....	18
Data Base.....	19
Graphics.....	19
Sounds.....	19
Goals Summary.....	19
Accomplished goals.....	19
Used Technologies.....	19
Networking level: Torque Network Library (OpenTNL).....	19
Design Fundamentals.....	19
Bandwidth conservation.....	20
Coping with Packet Loss.....	20
Strategies for Dealing With Latency.....	21
Architectural Overview.....	22
The Platform Layer.....	22
The NetBase Layer.....	22
The BitStream and PacketStream classes.....	22
The NetInterface and NetConnection Layer.....	22
The Event Layer - EventConnection, NetEvent and Remote Procedure Calls.....	23
Ghosting - GhostConnection and NetObject.....	23
Encryption and TNL.....	23
Useful Classes.....	24
Debugging and Error Handling.....	24
Remote Procedure Calls (RPC).....	24
3D Engine OGRE.....	26
Features.....	26
UML Overview.....	28
SceneManagers.....	28

Selecting a Scene Manager .....	28
Octree Scene Manager .....	28
Terrain Scene Manager .....	29
Nature Scene Manager (ogreaddons) .....	29
Paging Scene Manager (ogreaddons) .....	29
BSP Scene Manager .....	30
Dot Scene Manager (ogreaddons) .....	30
Entities.....	30
SceneNodes.....	30
Coordinates and Vectors.....	31
Vector.....	31
Representation of a vector.....	31
Length of a vector.....	32
Vector equality.....	32
Vector addition and subtraction.....	32
Scalar multiplication.....	33
Unit vector.....	34
Dot product.....	34
Cross product.....	34
Scalar triple product.....	35
Cameras.....	35
Viewports.....	35
Shadows.....	36
Lights.....	36
Sky.....	36
SkyBoxes .....	36
SkyDomes .....	36
SkyPlanes .....	36
Which to Use? .....	37
FrameListeners.....	37
KeyListener.....	37
MouseListener.....	37
MouseMotionListener.....	37
Implementation.....	37
Requirements.....	37
GNU/Linux.....	37
MS Windows.....	38
Screenshots.....	39
Networking Layer.....	40
3D Interface Layer.....	41
Suggestions for further work.....	41
Coordinates in Hexagon based tile maps.....	41
The mathematical structure of an hexagon.....	42
Converting array coordinates to pixel coordinates.....	43
Converting pixel coordinates to array coordinates.....	43
Path finding.....	45
Breadth-first search.....	48
Bidirectional breadth-first search.....	48
Dijkstra's algorithm.....	49
Depth-first search.....	50
Iterative-deepening depth-first search.....	50
Best-first search.....	51
A* Search.....	51
References.....	53
Other Resources.....	54

## Games Definitions

### *Massively Multiplayer Online Game (MMOG)*

**[Wikipedia<sup>1</sup> definition]** A massive multiplayer online game (MMOG) is a type of computer game<sup>2</sup> that enables hundreds or thousands of players to simultaneously interact in a game world they are connected to via the Internet. Typically this kind of game is played in an online, multiplayer-only persistent world.

There are a number of factors shared by most MMOGs that make them different from other types of computer games. MMOGs create a persistent universe where the game continues playing regardless of whether or not anyone else is. Since these games strongly or exclusively emphasize multiplayer gameplay, few of them have any significant single-player aspects or client-side artificial intelligence. As a result, players cannot "beat" MMOGs in the typical sense of single-player games. You will find lots of NPCs<sup>3</sup> and mobs<sup>4</sup> in most MMOGs who give out quests or serve as opponents.

There is some debate if a high head-count is the requirement to be a MMO. Some say that it is the size of the game world and its capability to support a large number of players that should matter. For example, despite technology and content constraints, most MMOGs can fit up to a few thousand players on a single game server at a time. Given technology development online multiplayer may eventually become MMOs as well as the average server size increases. However, by then the benchmark may have increased as well and MMO's will host many times even that. Alternatively, if the defining characteristic of MMOs is that all the players must be in a single-world where they can interact, online games with a highly fragmented un-connected sever base are not MMOs.

To support all those players, MMOGs need large-scale game worlds. In MMOGs, large areas of the game are interconnected within the game such that a player can traverse vast distances without having to switch servers manually.

There are few more common differences between MMOGs and other online games. Most MMOGs charge the player a monthly fee to have access to the exclusive servers. The game state in a MMOG rarely resets; what the player earned yesterday is with them still today. MMOGs often feature in-game support for clans and guilds, such as the ability to manage an association with in-game tools.

---

<sup>1</sup> Free online encyclopedia: <http://en.wikipedia.org/>

<sup>2</sup> Formally, a computer game is a game composed of a computer-controlled virtual universe that players may interact with in order to achieve a goal (or set of goals). A video game is a computer game where a video display is the primary feedback device.

<sup>3</sup> A non-player character or non-playable character is a fictional character in a role-playing game whose role is generally created and performed by the gamemaster.

<sup>4</sup> Abbreviation for mobile object.

## ***Massively Multiplayer Online Role-playing Game***

**[Wikipedia definition]** A massive(ly) multiplayer online role-playing game or MMORPG is a multiplayer computer role-playing game that enables thousands of players to play in an *role* evolving virtual world at the same time over the Internet. MMORPGs are a specific type of massive(ly) multiplayer online game (MMOG).

### **Academic attention**

**[Wikipedia]** MMORPGs have begun to attract significant academic attention, notably in the fields of economics and psychology. Edward Castronova<sup>5</sup> specializes in the study of virtual worlds (MUDs, MMOGs, and similar concepts). Most of his writings, including "Virtual Worlds: A First-Hand Account of Market and Society on the Cyberian Frontier" (2001), have examined relationships between real world economies and synthetic economies.

With the growing popularity of the genre, a growing number of psychologists and sociologists study the actions and interactions of the players in such games. One of the more famous of these researchers is Sherry Turkle. Nicholas Yee has surveyed thousands of MMORPG players over the past few years in studying the psychological and sociological aspects of these games.

## ***Real-time Strategy (RTS)***

**[Wikipedia definition]** A real-time strategy (RTS) game is a type of computer strategy game<sup>6</sup> which does not have "turns" like conventional turn-based strategy video or board games. Rather, game time progresses in "real time": that is, it is continuous rather than turn-by-turn.

Because of the generally faster-paced nature (and the usually shallower learning curve), RTS games have surpassed the popularity of conventional turn-based strategy computer games. In the past some traditional strategy gamers regarded RTS games as "cheap imitations" of turn-based games, arguing that RTS games had a tendency to devolve into "clickfests", in which the player who was faster with the mouse generally won, because they could give orders to their units at a faster rate. Real-time strategy enthusiasts counter that micromanagement involves not just fast clicking but also the ability to make sound tactical decisions under time pressure. It is noteworthy, however, that due to the games being shorter because of the faster pace of the game and absence of turn switching pauses, RTS games are far more suitable for Internet play than turn-based games; this is indubitably an important reason for their popularity.

---

<sup>5</sup> Edward Castronova is Associate Professor of Telecommunications at Indiana University Bloomington as of fall 2004, previously Associate Professor of Economics in the College of Business and Economics at California State University, Fullerton.

<sup>6</sup> Strategy games are typically board games, video or computer games with the players' decision-making skills having a high significance in determining the outcome.

## Typical Game Components

A game is normally composed by these components:

- **3D Engine**

It is in charge of the 3D graphics, and it gives more facilities than working directly with OpenGL. It allows you to load 3D models done with extern programmes like 3D Studio, Blender or Wings3D.

Available technologies:

- OGRE (<http://www.ogre3d.org/>) / Axiom3D (<http://www.axiom3d.org/>)  
It is possible to work with C++ or Python, even C# if we use Axiom3D.
- Crystal Space 3D (<http://crystal.sourceforge.net/>) for C++
- Xith 3D (<http://xith.org/>) for Java.
- jMonkey (<http://www.jmonkeyengine.com/>) for Java.
- LWJGL (<http://www.lwjgll.org/>) for Java.

- **Artificial Intelligence**

Usually it is used a script language integrated in the code in order to have more flexibility for AI. So, every time you want to change the AI it is not necessary to recompile all the project, the scripts are loaded dynamically. Several AI algorithms are used to improve the system behaviour.

Available technologies:

- Lua <http://www.lua.org/>
- AngelScript <http://www.angelcode.com/angelscript/>
- GameMonkey <http://www.somedude.net/gamemonkey/>
- Python
  - <http://docs.python.org/ext/ext.html>
  - <http://www.boost.org/libs/python/doc/index.html>
  - <http://www.swig.org/>
- GALib, A C++ Library of Genetic Algorithm Components (<http://lancet.mit.edu/ga/>)

- **Networking library**

It is important to use libraries for minimizing latency and needed bandwidth. There are libraries that use UDP as base protocol and add an extra layer that guarantees reliability and correct order.

- OpenTNL <http://www.opentnl.org/>
- Zig <http://zige.sourceforge.net/>

- **Physics**

There are libraries for detecting collisions.

- ODE <http://ode.org/>

- **Sound**

It is possible to use several libraries for playing sounds.

- OpenAL with 3D sound <http://www.openal.org/>
- FMod <http://www.fmod.org/>

# Game Design

## *Design Patterns*

BOOK: Design Patterns (ISBN 0201633612)

Authors: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Gang of Four, GoF)

**[Wikipedia definition]** In software engineering, design patterns are standard solutions to common problems in software design. The phrase was introduced to computer science in 1995 by the text Design Patterns: Elements of Reusable Object-Oriented Software. The scope of the term remained a matter of dispute into the next decade. Algorithms are not thought of as design patterns, since they solve computational problems rather than design problems. Typically, a design pattern is thought to encompass a tight interaction of a few classes and objects.

Design patterns can speed up the development process by providing almost ready-made solutions that have been used earlier and proved to be efficient.

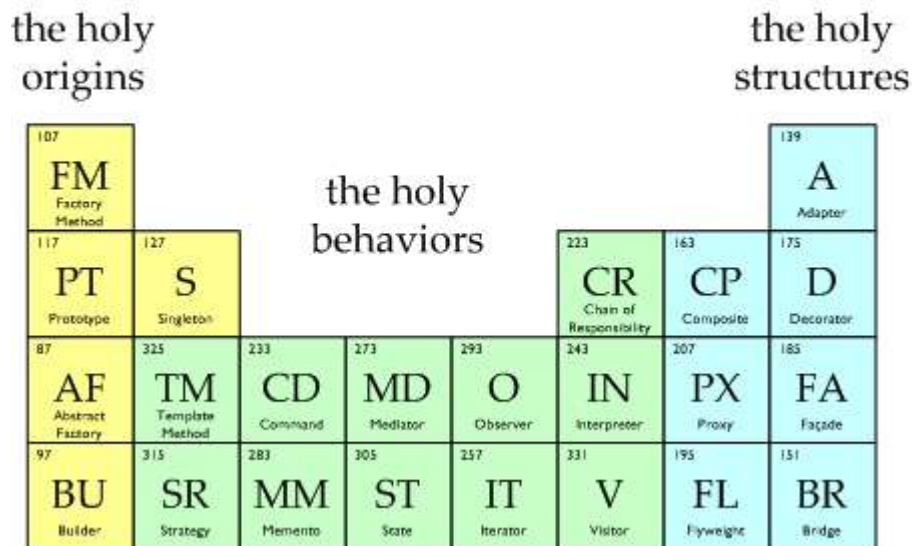
Although many of the problems frequently encountered in software design have existing solutions, solutions can be difficult to adopt due to the need to understand their details. Design patterns address this problem by being general; a solution documented in the format of a design pattern can be understood without knowledge of the specific details involved. Moreover, the standardized naming of design patterns makes communication among developers easier.

Commonly used design patterns also have the potential of being revised and improved over time, and thus are more likely to perform better than home made designs.

Design patterns can be classified based on multiple criteria, the most common of which is the basic underlying problem they solve. According to this criterion, design patterns can be classified into various classes, some of which are:

- Creational Patterns
  - Abstract factory pattern
  - Builder
  - Factory method pattern
  - Prototype
  - Singleton
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioral Patterns
  - Chain of responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template method
  - Visitor

## The Sacred Elements of the Faith



In order to get a good game design it is necessary to use the design patterns that best suit the application. There are also researches related to *Game Design Patterns*<sup>7</sup>:

PAPER: Game Design Patterns  
Authors: Staffan Björk, Sus Lundgren, Jussi Holopainen

PAPER: Design Patterns for Games  
Authors: Dung ("Zung") Nguyen and Stephen B. Wong

But they do not describe game's internal, just abstract ideas about the different objective of a game. So it does not give us a functional schemas for developing games. Gamma's Generic Design Pattern seems more useful for any kind of developing.

### Initial Documentation

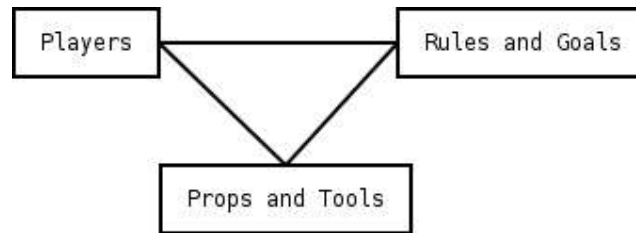
PAPER: A model to support the design of Multiplayer Games  
Authors: José Pablo Zagai, Miguel Nussbaum, Ricardo Rosas

It is necessary to determine:

- Game structure
- Scope
- Tokens (items with which a person interacts while playing)
- Interactivity
- Implementation details

<sup>7</sup> Game Design Patterns, <http://civ.idc.cs.chalmers.se/projects/gamepatterns/>

In order to do so we can use the Simple Model of a Game which divides the game in three main components:



We have to define the different parts of the diagram and its relationships:

- Rules and Goals
  - Rules: regulate the development of the game
  - Goals: objective that the player shall pursue
- Props and Tools
  - Prop: element that is used for decorative purposes
  - Tool: element used by the player
- Players
  - To what extent do the rules affect social interaction?
  - To what extent do the props and tools affect social interaction?

Interaction can happen in two different flavours, so one of them must be chosen:

- Natural Interaction (spontaneous)
- Stimulated Interaction (necessary for the game)

Finally it is important to determine some other general concepts:

- Competition: How do players compete in order to reach the objective?
- Cooperation: It is needed/possible to cooperate in order to reach the objective?
- Synchronous or asynchronous game? Must all the players be online at the same time to play?

## Technical Decisions

### Networking

PAPER: Aspects of Networking in Multiplayer Computer Games  
 Authors: Jouni Smed, Timo Kaukoranta, Harri Hakonen

MMOG are network oriented game, so we have some resource limitations: network bandwidth, network latency, and host processing power for handling the network traffic.

- Bandwidth refers to the transmission capacity of a communication line such as a network.
- Networking latency indicates the length of time (or delay) that incurs when a message gets from one designated node to another. In addition, the variance of latency over time (i.e., jitter) is another feature that affects interactive applications. Latency cannot be totally eliminated.
- Computational Power: Network traffic handling puts additional computational strain on the computer running a distributed system.

Compensatory Techniques for reducing resource requirements:

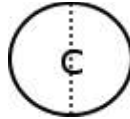
- Message Compression and Aggregation: Send several data together and compressed.



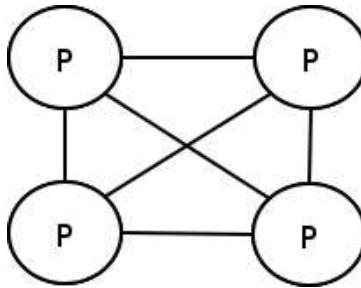
- Interest Management: It allows the nodes to express interest in only the subset of information that is relevant to them. They aim at reducing the number of transmitted messages by specifying the interested received.
- Dead Reckoning: It is based on a navigational technique of estimating one's position based on a known starting point and velocity. The same idea can be applied to predicting the data from the other nodes, which allows to prolong the interval of message transmissions and abolish the network latency at the cost of data consistency.

### Communication Architectures

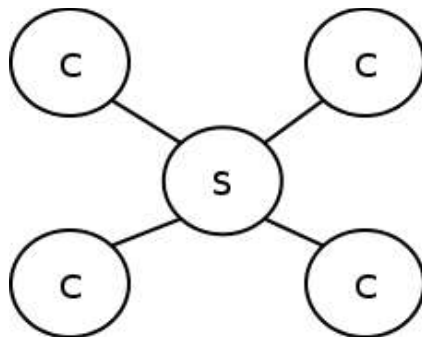
- Single node: Split screen for having several players in one computer. It is not the MMOG case.



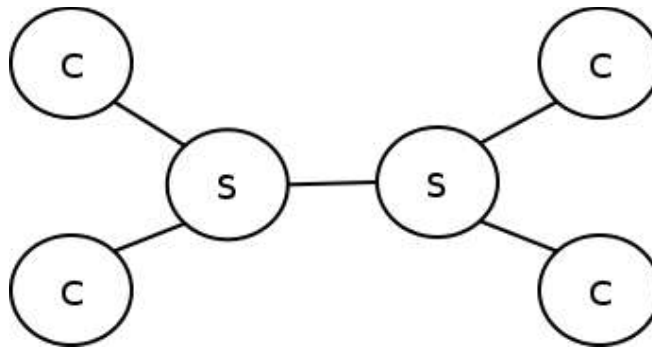
- Peer-to-Peer: We have a set of equal nodes connected. Since no node is more special than the other, they must be connected to each other. It does not scale up easily due the lack of structure.



- Client-Server: All communication is handled through this server node, while the other nodes remain in the role of a client. The server becomes the critical part, if it cannot keep up with the network traffic.



- Server-Network: Several interconnected servers. This provides better scalability but increases the complexity of handling the network traffic.



## Problem of Server-Network: Scalability

**[Wikipedia definition]** In telecommunications and software engineering, scalability indicates the capability of a system to increase performance under an increased load when resources (typically hardware) are added. Scalability is a highly significant issue in electronics systems, database, routers, networking, etc. and implies performance. A system whose performance improves after adding hardware, proportionally to the capacity added, is said to be a scalable system.

Measurements:

- Speedup =  $S = T(1) / T(n)$
- Efficiency =  $E = (S / n) * 100$

\*  $n$  = number of computers, the utopic target is  $S = n$  or  $E = 100\%$ .

In a communication architecture like Server-Network is difficult to get a scalable system because of Communications + Synchronization + Management between servers give us some Overhead which makes increase  $T(n)$ . Also it is not possible to do everything in parallel because normally there are a lot of parts of programs that need to be computed serially.

## Security

**[Wikipedia definition]** Computer security is the effort to create a secure computing platform, designed so that agents (users or programs) can only perform actions that have been allowed. This involves specifying and implementing a security policy. The actions in question can be reduced to operations of access, modification and deletion. Computer security can be seen as a subfield of security engineering, which looks at broader security issues in addition to computer security.

In MMOG that Security means:

- Protecting sensitive data with encryption.
- Providing a fair play field (avoiding cheats).

**[Wikipedia definition]** Cheating is a common problem in multiplayer, online computer games. While there have always been cheat codes and other ways to make single player games easier, most game developers focus on making multiplayer games "fair". With the release of the first popular Internet multiplayer games (Quake being one of the first) cheating took on new dimensions. Previously it was rather easy to see if the other players cheated, as most games were played on local networks. The Internet changed that.

In online games, cheating is generally regarded as modifying the game experience in some way in order to gain an advantage over

the other players. Because almost all games today allow some level of reconfiguration, the amount of reconfiguration which constitutes "cheating" differs between different games and communities. Changing the keyboard layout to make it easier to use is for example generally accepted. But issues such as changing in-game player models and textures, or modifying the brightness or gamma in order to make it easier to see in dark areas are often contended. There are hundreds of ways to make things easier in an unfair way, and it is practically impossible to block all of them without complete control over the computers the games are run on.

Usually included in this concept of cheating is the exploiting of existing bugs or unintended gameplay known in leet as "sploitiz". Gamers are divided on whether exploiters are cheaters, though most agree that especially unfair exploits are cheats. Exploiters have argued that it requires no changes to exploit bugs and thus anyone can do it. They also say that it is not obvious that their style of play is unintended. Opponents disagree and might argue that the exploits ruin the game the same way a cheat does and that the only reason it is possible to do them is because they have not been fixed. Duping (commonly a bug) can ruin a synthetic economy, is rarely intended, and therefore is usually called a cheat.

**The problem:** There are many aspects of cheating in online games which make the creation of a system to stop cheating very difficult.

"Never trust the client" is a common maxim among multiplayer game developers that summarizes the problems client-server games have. It argues that programmers should assume that information sent to the client game will be known by that player, regardless of whether or not the player should know that information. For example, the server might tell a client in an FPS game that a player is hiding behind a door and cannot be seen, but a wallhack cheat can reveal the player. Similarly, data from the client might indicate that the client teleported from one side of the map to another for no reason (possibly a change made the game's data). The server is responsible for sending only the necessary information and for maintaining the game's continuity.

## Packet and Traffic Tampering

A cheater can turn on a proxy program in its client computer and:

- Modify the information being transmitted (e.g. Packet replay: transmitting more steps/shoots per second).
- Suppress the packets containing damage, so the cheater becomes invulnerable.

The user can try to change byte order in a packet and observe the effects in order to break the control. A partial solution can be using checksums (MD5 is well tested and fast enough for real-time games). But in any case cheaters can reverse engineer the checksum algorithm or they can attack with packet replay.

Random numbers can also be used to modify the packets so that even identical packets do not appear the same. Dissimilarity can be further induced by adding a variable amount of junk data to the packets, which eliminates the possibility of analysing their contents by the size. It does not avoid completely the possibility of reverse engineering and if the client is Open Source, anybody can have a look to the source code and learn how the protocol is working, so this kind of protection is worthless.

The best protection to these attacks is designing a good server model with a big number of checks on the received client data.

## Information Exposure

A cracked client software may allow a cheater to gain access to the replicated, hidden game data (e.g., the status of other players.)

In a centralized architecture, an obvious solution is to utilize the server, which can check whether a client issuing a command is actually aware of the object with which it is operating. For example, if a player has not seen the opponent's base, he cannot give an order to attack it--unless he is cheating. When the server detects cheating, it can drop out the cheating client.

So the communication architectures Server-Client and Server-Network are the most suitable for avoiding cheats.

## Design Defects

If the clients are designed to trust each other, the game is unshielded from client authority abuse.

Although this problem can be tackled by using checksums to ensure that each client has the same binaries, it is more advisable to alter the design so that the clients can issue command requests, which the server puts into operation.

**Efficiency versus security:** The more of the game code that is run on the server, the more secure the game will be generally, as the server's operator has control over what happens. However, a game server has limited bandwidth and limited resources, which makes it necessary to distribute code to the clients. It's a tradeoff between security and usability, where going too far either way might break an otherwise great game.

## Artificial Intelligence

PAPER: Human-like Behaviour in Real-Time Strategy Games: An Experiment With Genetic Algorithms

Authors: Fredrik Olofsson, Johan W. Andersson

PAPER: Using Genetic Algorithms for Game AI

Author: Greg James

### *Classical techniques and topics*

- Path finding: At the low level, determining how to move multiple characters across novel and possibly dynamic terrain is a problem central to many of the most popular game titles. e.g. A\* algorithm
- Search Tree: By knowing the state of the game at all times, we may derive consecutive game states from the current game state. This forms a sort of tree where each node, connected by edges, represents a game state. The number of nodes possible depends on the number of possible game states. In chess for example, the average branching factor is approximately 35. For real time games it is not useful unless for little problems with few states/branches.
- Finite state machines: They are simple directed and discrete graphs consisting of numerous states (player modes). That is, exactly one state can be occupied at any time. By transition the player moves from one state to another. For example, if the computer player in an FPS game is in the mode "idle" and the human opponent crosses its field of vision within a certain distance, a transition from mode "idle" to mode "hunt" could be made, and the computer player will then display another behaviour. The benefits of using finite state machines are its simplicity and the use of minimal resources needed to make one useful.
- Fuzzy Logic: To make computer players more interesting and flexible than they are when using other AI techniques, many have used what is known as fuzzy logic. Instead of using the two discrete states of true and false, as in normal logic, fuzzy logic uses real-value numbers. These real-value numbers usually indicates a "desire" of doing something; i.e. enforcing certain behaviours. This could be a distance to move, chance of attacking a foe etc. By not having discrete states, it is also possible to let the computer player implement many strategies at the same time, only to differ to what extent they're applied.
- Asymmetric Rules: It is important to develop several levels of difficulties. It can be in two ways:
  - Doing a very good IA and eliminating parts for every lower level
  - AI Cheating (not related to client/player cheating)

### *Genetic Algorithms*

Genetic algorithms are one of a variety of AI techniques that have been extensively explored by academics but have yet to push their way into game development. However, they offer opportunities for developing interesting game strategies in areas where traditional game AI is weak.

Genetic algorithms (GAs) are one of a group of random walk techniques. These techniques attempt to solve problems by searching the solution space using some form of guided randomness. Another technique of this type is simulated annealing.

The main issue with genetic algorithms is that there are few jobs that GAs can do better than a heuristic-based search. Generally speaking, if you have some ideas of how to solve your problem, you're probably better off implementing those ideas that you are turning your problem over to a GA, which is relying on randomness. But there are two issues in game development that may warrant their use: noise and data access.

By noise we mean that the evaluation of a given candidate solution may vary randomly.

Data access means that you may not have the ability to get at the information you want into order to make good decisions with other more obvious types of AI. For example, when humans play StarCraft<sup>8</sup> they usually set up some sort of defence for their base depending on what type of attack they're expecting. Any heuristic algorithm needs to have an analysis of the defence in order to produce a recommended offence, and writing that programmatic analysis could be extremely difficult.

By using genetic algorithms an opponent capable of learning and adapting could be realised. This could be a way to make the opponent more human-like and at the same time address the one-time syndrome (once solved, the interest fades).

Traditional game AI has usually involved fighting with the graphics and sound programmers for scarce CPU time and resources. Genetic algorithms are *computationally expensive*, and work better the more resources you can throw at them. Larger populations and more generations will give you better solutions. This means that GAs are better used *offline*. One way of doing this is by doing all of the GA work in-house and then releasing an AI tuned by a GA.

It is also possible to use Genetic Algorithms during playing real-time games, but we have to start with a good AI base maybe with a offline trained one or with an static hard coded base. Anyway, it is not a popular technique nowadays.

---

<sup>8</sup> StarCraft (SC) is a real-time strategy computer game produced by Blizzard Entertainment in 1998. The game is similar to Blizzard's previous hit Warcraft II, but has a science fiction setting.

## General Design Questions

In order to design a complete commercial MMOG, it is necessary to find answer to an important group of questions that I have grouped in several themes and that you kind find in this section. At the end of the design process, most of the question must have a clear answer.

### *Connection Issues*

- Must personal data be encrypted?
- What data must be encrypted? Login, password, credit card, ...?
- TCP or UDP? Reliable or Unreliable UDP?
- “Non-blocking sockets, non-threaded” (easier to implement/debug, slower response) or “Blocking sockets, threaded” (faster response, improvements with multiprocessor, hard to implement, lot of synchronize work)?
- How can be ensured that the packets you're receiving are really from your client software?
- How to avoid Denials of Services (DoS)?
- Bandwidth needed?
- How can Latency be reduced?
- What will be done, both at the client and server ends, if the connection drops or packets are lost? Control it using timeouts?
- What conventions will be used in the networking protocol? C-style null terminated strings, or you could go with Pascal-style strings where the length is prepended to the string?

Example:

**Offset 0:** 1 byte denoting the command transmitted.

**Offset 1:** 2 bytes, the length of the data transmitted.

**Offset 3:** variable length, the body of the message.

What to do with fixed messages? Hybrid method?

### *Operating Platform Issues*

- How many players do we expect (total subscribers and concurrent players)?
- How is going to be spread the load over multiple servers?

Separate physics from IA?

Multiple mirrored servers?

Map divided into different servers?

- Check every received data from outside (IDs, objects, ...), avoid buffer overflows (fixed buffers) or cheats. Log username/IP/time/date/description when a violation happens, if the violation occurs frequently from the same username it can be a cheater, otherwise it can be a problem with the client. All the information must be stored in the server, just needed things must be send to client.

- What will be the Internal network topology?
- How can be the world restored to a state as close as possible to before a hardware/software failure?

Which parts must be backed up?

Shall we use RAID Systems?

- Database synchronization, should we use atomic operations?

What database should we use? PostgreSQL? MySQL? Firebird?

## ***Technologies issues***

- What do we have to cache in the server and in the client?
- What forms part of the client? Graphic data?
- How to deal with updates? Stop every server?

Dynamic updates?

What should we do if we are modifying a resource that is used by players (e.g. we are removing it)?

How can we make a dynamic update in every server?

How can we make a dynamic update in the client?

- What kind of data is coming from the client?

Relative operation better than absolute operations (e.g. positioning)?

Do we have to control the key pressing rate of the client? Server must control movements guided by time clock.

Do we have to look for debuggers or applications that alter memory?

Checksum data files in order to check that they are not modified?

- How can we keep synchronized server and clients?

How can we keep synchronized servers among them?

## ***Business Issues***

- Structure: One firm for developing and one for running the game?
- Maintenance costs: Hardware, electricity, Internet connection, employers... for running the game.
- Content development: How will new content for the game be developed?

Keep the same programmers that develop the game or hire new people so the old ones can move to another project?

Create a community to contribute to the game?

- Is it better to make an Open Source client? Cheated clients must be detected by the server but community can improve client.

- Revenue streams: Will it be based in purely subscriptions fee?

Will the client be charged by the initial client portion of the game?

Will we have different levels of subscriptions?

Will we have unlockable content for players paying more?

- Middleware: Will be used software to support several servers? Grid computing (e.g. Butterfly.net)?



## Project

### MMOG Idea

In the classical computer games which are available today in this market you create a character in the world the game happens in and play with it. The goal of the game and the character usually involves killing monsters, finding treasures and in this way it gains experience and becomes more powerful.

What sets the proposed MMOG apart from other games is that the main focus will be on towns, lands and buildings instead of the character that is being played, in consequence this MMOG is not like the classical games.

The game can take place in an imaginary world in the middle ages where magic, gods, and monsters exists. In the beginning the players chooses what path he wishes his character to take (players interact with the system assuming a role), for example:

- Military
- Church
- Running a store

After this, the players starts with his own house in the town which he chooses along with a family owned property which could be one of the churches in the town, a tavern, blacksmith or whatever the player chooses. The game then emphasizes on the growth of your power, your dynasty, trade and exploration. So the game has to have a living and thriving economy.

It tries to emphasize on an aspect of gaming than is usual for games of this type and try to appeal more to the female community of players. There are social researches about MMOG that says “What women are finding so interesting about these games is that they provide a sense of community and social structure that you don't see in other games”.

PAPER: Social Networking in Massively Multiplayer Online Games Authors: Mikael Jakobsson, T.L. Taylor
--

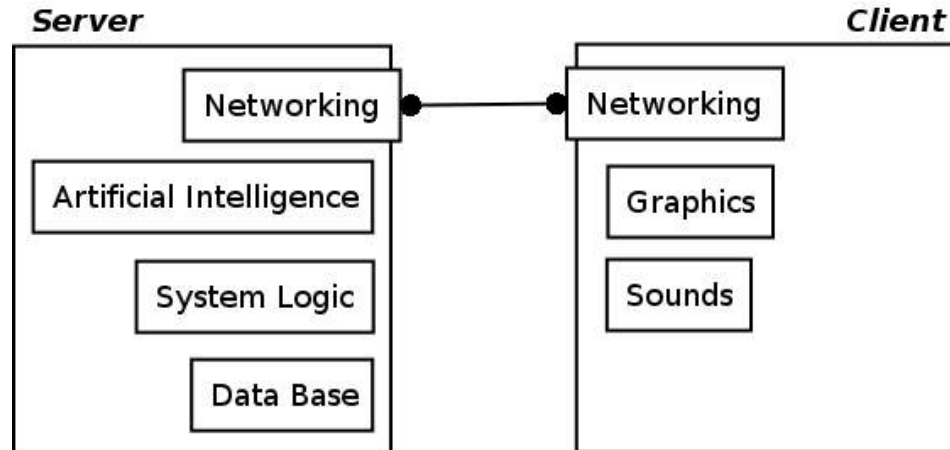
Article: Men are from Quake, Women are from Ultima <sup>9</sup> Emily Laber (NY Time article)
--

---

<sup>9</sup> Article's link: <http://www.nytimes.com/2001/01/11/technology/11ROLE.html?pagewanted=all>

## Goals

Design a potential MMORPG/MMOG (Real-Time Strategy view) using Design Pattern in order to get an elegant model, easy to maintain and improve. It is important to try to apply all the techniques and researches done in order to make a secure system (for example avoiding cheats).



The design have to give solutions for implementing each of the parts of the above diagram, it is needed to try the different existing technologies like frameworks and libraries so it would be possible to see if with already existing components we can build a good MMOG. It is important to try to reuse as much code as possible.

The communication architecture will be Server-Client so the server side will receive special attention as it is the kernel of the system. Graphics and Sounds will be the less important part of the design.

## Networking

Using the current networking libraries, try to identify the kind of data that must be transmitted between Server and Client in order to have a good efficiency. Security is also a point to identify in order to avoid cheating.

## Artificial Intelligence

In the game there will be different important roles like Politics one, they have powers to raise leasing price, sales tax, etc... also we can have other roles like Judge, Bishop, Building master, etc... If there is a vacant in one of this roles, AI will take control of this position until a real player can assume the role.

It can be interesting to design a *genetic algorithm* in order to get a human behaviour of this virtual politics, otherwise a classical approach can also be valid.

Graphically the system will show monster, workers and other moving elements that can be controlled by System Logic. If any of these parts need to follow a path to go from one point to another, then the AI will calculate how to do it using path finding algorithms.

## System Logic

System must control all the market economy with different prices depending on supply and demand in the country and towns. Workers will also be a part of the system, so they must be represented.

System logic will be in charge of coordinating all the elements of the server-side.

## Data Base

MMOG are based in static state worlds, so the information must be saved in persistent storage like Data Base. It is important to select a suitable Data Base for this kind of games and decide the data that will be saved (defining an structure).

## Graphics

This is one of the less important parts of the project. It is possible to see what kind of frameworks can be used for developing a MMOG.

## Sounds

With Graphics, Sounds is another of the less important parts of the project. It is possible to test the different available libraries in order to decide which can be the best for our purpose.

## Goals Summary

- Design a potential MMORPG/MMOG (Real-Time Strategy view) using Design Pattern in order to get an elegant model, easy to maintain and improve. Server side will have special importance because it is the kernel of the system. The information found in this Final Project Proposal will be the base for the design (Networking and AI are the most important parts).
- Try to implement some part using as many libraries as possible in order to reuse code. The objective is try to see if the existing technologies are good enough for developing a MMOG of this characteristics.

## Accomplished goals

The basic part of the total project has been designed and implemented successfully:

- Networking level has been designed and implemented using OpenTNL. Server and Client sides are completely functional, they are optimized for minimum bandwidth usage and strong encryption communications.
- 3D Enviroment has been designed and implemented using OGRE. The user is capable of moving through a simple 3D world and controlling its character.
- Menu system for user interaction implemented using CEGUI. User can access to login screen and character selection window, the real functionality is not implemented (password is not validated and there are not different characters).

See implementation section for further details.

## Used Technologies

Maximize reusability of code has been one of the main objective. For each of the different specified layers in the Game Design section, I have selected the most powerful available technology under a Free Licence (e.g. GPL, LGPL). In this section we will see the designs of these specific technologies and how they have been used in the project.

### ***Networking level: Torque Network Library (OpenTNL)***

[Information extracted from [www.opentnl.org](http://www.opentnl.org)]

## Design Fundamentals

The Torque Network Library was designed to overcome, as much as possible, the three fundamental limitations of network programming - high packet latency, limited bandwidth and packet loss. The following strategies were identified for each of the limitations:

### ***Bandwidth conservation***

Bandwidth conservation in multiuser simulations is of premium importance, not only for clients that may be connected over very low bandwidth transports, but also for servers whose operators have to pay for bandwidth usage. Realizing this, the following techniques are used to conserve bandwidth as much as possible:

**Send static data once, or not at all:** Often in a networked application, a server will transmit some amount of static data to a client. This data may be initialization parameters for a 3D world, a client's name, or some object state that is immutable for objects of a given type. TNL provides direct facilities for caching string data (client names, mission descriptions, etc), sending a given string only once and an integer representation thereafter. The Torque Game Engine also shows how simulations can cache common object instance data in DataBlock objects, which are transmitted only when a client connects.

**Compress data to the minimum space necessary:** When conserving bandwidth, every bit counts. TNL uses a utility class call BitStream to write standard data types into a packet compressed to the minimum number of bits necessary for that data. Boolean values are written as a single bit, integer writes can specify the bit field width, and floating point values can be specified as 0 to 1 compressed to a specified bit count. The BitStream also implements Huffman compression of string data and compression of 3D positions and surface normals.

**Only send information that is relevant to the client:** In a client/server simulation, the server often has information that is not relevant at all to some or all of the clients. For example, in a 3D world, if an object is outside the visible distance of a given client, there is no reason to consume valuable packet space describing that object to the client. The TNL allows the application level code to easily specify which objects are relevant, or "in scope" for each client.

**Prioritize object updates:** Because bandwidth is limited and there is generally a much greater amount of data a server could send to a given client than it has capacity for, the TNL implements a very fine-grained prioritization scheme for updating objects. User code can determine the policy by which objects are judged to be more or less "important" to each client in the simulation, and objects with more importance are updated with a greater frequency.

**Only update the parts of an object that have changed:** Often in a networked simulation, not all object state is updated at the same time - for example, a player in a 3D action game may have state that includes the player's current position, velocity, health and ammunition. If the player moves, only the position and velocity states will change, so sending the full state, including the health and ammunition values would waste space unnecessarily. The TNL allows objects individual objects to have state that are updated independently of one another.

### ***Coping with Packet Loss***

In any networked simulation, packet loss will be a consideration - whether because of network congestion, hardware failure or software defects, some packets will inevitably be lost. One solution to the packet loss problem is to use a guaranteed messaging protocol like TCP/IP. Unfortunately, TCP has some behavioural characteristics that make it a poor choice for real-time networked simulations.

TCP guarantees that all data sent over the network will arrive, and will arrive in order. This means that if a data packet sent using TCP is dropped in transit, the sending host must retransmit that data before any additional data, that may have already arrived at the remote host, can be processed. In practice this can mean a complete stall of ANY communications for several seconds. Also, TCP may be retransmitting data that is not important from the point of view of the simulation - holding up data that is.

The other possible protocol choice would be to use UDP for time critical, but unguaranteed data, and use TCP only for data that will not hold up the real-time aspects of the simulation. This solution ends up being non-optimal for several reasons. First, maintaining two communications channels increases the complexity of the networking component. If an object is sent to a client using the guaranteed channel, unguaranteed messages sent to that object at a later time may arrive for processing before the original guaranteed send. Also, because the unguaranteed channel may lose information, the server will have to send more redundant data, with greater frequency.

To solve these problems, TNL implements a new network protocol that fits somewhere between UDP and TCP in its feature set. This protocol, dubbed the "Notify" protocol, does not attempt to hide the underlying unreliability of the network as TCP does, but at the same time it provides more information to the application than straight UDP. The notify protocol is a connection-oriented unreliable communication protocol with packet delivery notification. When a datagram packet is sent from one process, that process will eventually be notified as to whether that datagram was received or not. Each data packet is sent with a packet header that

includes acknowledgement information about packets the sending process has received from the remote process. This negates the need for separate acknowledgement packets, thereby conserving additional bandwidth.

The notify protocol foundation allows the TNL to provide a rich set of data transmission policies. Rather than simply grouping data as either guaranteed or unguaranteed, the TNL allows at least five different categorizations of data:

**Guaranteed Ordered data:** Guaranteed Ordered data are data that would be sent using a guaranteed delivery protocol like TCP. Messages indicating clients joining a simulation, text messages between clients, and many other types of information would fall into this category. In TNL, Guaranteed Ordered data are sent using Event objects and RPC method calls. When the notify protocol determines that a packet containing guaranteed ordered data was lost, it requeues the data to be sent in a future packet.

**Guaranteed data:** TNL processes Guaranteed data in a way similar to Guaranteed Ordered data, with the only difference being that a client can process Guaranteed data as soon as it arrives, rather than waiting for any ordered data to arrive that was dropped.

**Unguaranteed data:** Unguaranteed data is sent, and if the packet it is sent in arrives, is processed. If a packet containing unguaranteed data events is dropped, that data is not resent. The unguaranteed data sending policy could be used for information like real-time voice communication, where a retransmitted voice fragment would be useless.

**Current State data:** For many objects in a simulation, the client isn't concerned with "intervening" states of an object on the server, but only its current state. For example, in a 3D action game, if an enemy player moves from point A to B to C, another client in the simulation is only interested in the final position, C of the object. With the notify protocol, TNL is able to resend object state from a dropped packet only if that state was not updated in a subsequent packet.

**Quickest Delivery data:** Some information sent in a simulation is of such importance that it must be delivered as quickly as possible. In this situation, data can be sent with every packet until the remote host acknowledges any of the packets known to contain the data. Client movement information is an example of data that might be transmitted using this policy.

By implementing various data delivery policies, the TNL is able to optimize packet space utilization in high packet loss environments.

## ***Strategies for Dealing With Latency***

Latency is a function of the time-based limitations of physical data networks. The time it takes information to travel from one host to another is dependent on many factors, and can definitely affect the user's perceptions of what is happening in the simulation.

For example, in a client-server 3D simulation, suppose the round-trip time between one client and server is 250 milliseconds. If the client is observing an object that is moving. If the server is sending position updates of the object to the client, those positions will be "out of date" by 125 milliseconds by the time they arrive on the client. Also, suppose that the server is sending packets to the client at a rate of 10 packets per second. When the next update for the object arrives at the client, it may have moved a large distance relative to the perceptions of the client.

Also, if the server is considered to be authoritative over the client's own position in the world, the client would have to wait at least a quarter of a second before its inputs were validated and reflected in its view of the world. This gives the appearance of very perceptible input "lag" on the client.

In the worst case, the client would always see an out of date version of the server's world, as objects moved they would "pop" from one position to another, and each key press wouldn't be reflected until a quarter of a second later. For most real-time simulations, this behaviour is not optimal.

Because TNL is predominantly a data transport and connection management API, it doesn't provide facilities for solving these problems directly. TNL provides a simple mechanism for computing the average round-trip time of a connection from which the following solutions to connection latency issues can be implemented:

**Interpolation:** Interpolation is used to smoothly move an object from where the client thinks it is to where the server declares it to be over some short period of time. Parameters like position and rotation can be interpolated using linear or cubic interpolation to present a consistent, no "pop" view of the world to the client. The downside of interpolation when used alone is that it actually exacerbates the time difference between the client and the server, because the client is spending even more time than the one-way message time to move the object from its current position to the known server position.

**Extrapolation:** To solve the problem of out-of-date state information, extrapolation can be employed. Extrapolation is a best guess of the current state of an object, given a known past state. For example, suppose a player object has a last known position and velocity. Rather than placing the player at the server's stated position, the player object can be placed at the position extrapolated

forward by velocity times the time difference.

In the Torque Game Engine, player objects controlled by other clients are simulated using both interpolation and extrapolation. When a player update is received from the server, the client extrapolates that position forward using the player's velocity and the sum of the time it will use to interpolate and the one-way message time from the server - essentially, the player interpolates to an extrapolated position. Once it has reached the extrapolated end point, the player will continue to extrapolate new positions until another update of the object is received from the server.

By using interpolation and extrapolation, the client view can be made to reasonably, smoothly approximate the world of the server, but neither approach is sufficient for real-time objects that are directly controlled by player input. To solve this third, more difficult problem, client-side prediction is employed.

**Client-side prediction** is similar to extrapolation, in that the client is attempting to guess the server state of an object the server has authority over. In the case of simple extrapolation, however, the client doesn't have the benefit of the actual input data. Client-side prediction uses the inputs of the user to make a better guess about where the client-controlled object will be. Basically, the client performs the exact same object simulation as the server will eventually perform on that client's input data. As long as the client-controlled object is not acted upon by forces on the server that don't exist on the client or vice versa, the client and server state information for the object should perfectly agree. When they don't agree, interpolation can be employed to smoothly move the client object to the known server position and client-side prediction can be continued.

## Architectural Overview

The Torque Network Library is built in layers, each adding more functionality to the layer below it.

### ***The Platform Layer***

At the lowest level, TNL provides a platform independent interface to operating system functions. The platform layer includes functions for querying system time, sleeping the current process and displaying alerts. The platform layer includes wrappers for all of the C standard library functions used in the TNL.

The platform layer also contains the Socket and Address classes, which provide a cross-platform, simplified interface to datagram sockets and network addresses.

### ***The NetBase Layer***

The NetBase layer is the foundation upon which most of the classes in TNL are based. At the root of the class hierarchy is the Object base class. Every subclass of Object is associated with an instance of NetClassRep through a set of macros, allowing for instances of that class to be constructed by a class name string or by an automatically assigned class id.

This id-based instantiation allows objects subclassed from NetEvent and NetObject to be serialized into data packets, transmitted, constructed on a remote host and deserialized.

Object also has two helper template classes, SafePtr, which provides safe object pointers that become NULL when the referenced object is deleted, and a reference pointer class, RefPtr, that automatically deletes a referenced object when all the references to it are destructed.

### ***The BitStream and PacketStream classes***

The BitStream class presents a stream interface on top of a buffer of bytes. BitStream provides standard read and write functions for the various TNL basic data types, including integers, characters and floating point values. BitStream also allows fields to be written as bit-fields of specific size. An integer that is always between 3 and 9 inclusive can be written using only 3 bits using the writeRangedU32 method, for example.

BitStream huffman encodes string data for additional space savings, and provides methods for compressing 3D points and normals, as routines for writing arbitrary buffers of bits.

The PacketStream subclass of BitStream is simply a thin interface that statically allocates space up to the maximum size of a UDP packet. A routine can declare a stack allocated PacketStream instance, write data into it and send it to a remote address with just a few lines of code.

### ***The NetInterface and NetConnection Layer***

The NetInterface class wraps a platform Socket instance and manages the set of NetConnection instances that are communicating

through that socket. NetInterface manages the two-phase connection initiation process, dispatch of protocol packets to NetConnection objects, and provides a generic interface for subclasses to define and process their own unconnected datagram packets.

The NetConnection class implements the connected Notify Protocol layered on UDP. NetConnection manages packet send rates, writes and decodes packet headers, and notifies subclasses when previously sent packets are known to be either received or dropped.

NetInterface instances can be set to use a public/private key pair for creating secure connections. In order to prevent attackers from depleting server CPU resources, the NetInterface issues a cryptographically difficult "client puzzle" to each host attempting to connect to the server. Client puzzles have the property that they can be made arbitrarily difficult for the client to solve, but whose solutions can be checked by the server in a trivial amount of time. This way, when a server is under attack from many connection attempts, it can increase the difficulty of the puzzle it issues to connecting clients.

### ***The Event Layer - EventConnection, NetEvent and Remote Procedure Calls***

The EventConnection class subclasses NetConnection to provide several different types of data transmission policies. EventConnection uses the NetEvent class to encapsulate event data to be sent to remote hosts. Subclasses of NetEvent are responsible for serializing and deserializing themselves into BitStreams, as well as processing event data in the proper sequence.

NetEvent subclasses can use one of three data guarantee policies. They can be declared as GuaranteedOrdered, for ordered, reliable message delivery; Guaranteed, for reliable but possibly out of order delivery, or Unguaranteed, for ordered but not guaranteed messaging. The EventConnection class uses the notify protocol to requeue dropped events for retransmission, and orders the invocations of the events' process methods if ordered processing is requested.

Because declaring an individual NetEvent subclass for each type of message and payload to be sent over the network, with its corresponding pack, unpack and process routines, can be somewhat tedious, TNL provides a Remote Procedure Call (RPC) framework. Using some macro magic, argument list parsing and a little assembly language, methods in EventConnection subclasses can be declared as RPC methods. When called from C++, the methods construct an event containing the argument data passed to the function and send it to the remote host. When the event is unpacked and processed, the body of the RPC method implementation is executed.

### ***Ghosting - GhostConnection and NetObject***

The GhostConnection class subclasses EventConnection in order to provide the most-recent and partial object state data update policies. Instances of the NetObject class and its subclasses can be replicated over a connection from one host to another. The GhostConnection class manages the relationship between the original object, and its "ghost" on the client side of the connection.

In order to best utilize the available bandwidth, the GhostConnection attempts to determine which objects are "interesting" to each client - and among those objects, which ones are most important. If an object is interesting to a client it is said to be "in scope" - for example, a visible enemy to a player in a first person shooter would be in scope.

Each GhostConnection object maintains a NetObject instance called the scope object - responsible for determining which objects are in scope for that client. Before the GhostConnection writes ghost update information into each packet, it calls the scope object's performScopeQuery method which must determine which objects are "in scope" for that client.

Each scoped object that needs to be updated is then prioritized based on the return value from the NetObject::getUpdatePriority() function, which by default returns a constant value. This function can be overridden by NetObject subclasses to take into account such factors as the object's distance from the camera, its velocity perpendicular to the view vector, its orientation relative to the view direction and more.

Rather than always sending the full state of the object each time it is updated across the network, the TNL supports only sending portions of the object's state that have changed. To facilitate this, each NetObject can specify up to 32 independent states that can be modified individually. For example, a player object might have a movement state, detailing its position and velocity, a damage state, detailing its damage level and hit locations, and an animation state, signifying what animation, if any, the player is performing.

A NetObject can notify the network system when one of its states has changed, allowing the GhostConnections that are ghosting that object to replicate the changed state to the clients for which that object is in scope.

### ***Encryption and TNL***

The TNL has hooks in various places to use encryption when requested. To enable encrypted connections, the NetInterface must be assigned an instance of AsymmetricKey as a public/private key pair using NetInterface::setPrivateKey. The

NetInterface::setRequiresKeyExchange method may then be called to instruct the NetInterface that all incoming connection requests must include a key exchange.

Asymmetric keys can be constructed with varying levels of security. The TNL uses Elliptic Curve public key cryptography, with key sizes ranging from 20 to 32 bytes. AsymmetricKey instances can be constructed either with a new, random key of a specified size, or can be created from an existing ByteBuffer.

Once a secure connection handshake is completed, TNL uses the AES symmetric cipher to encode connected packets. The BitStream::hashAndEncrypt and BitStream::decryptAndCheckHash methods use an instance of SymmetricCipher to encrypt and decrypt packets using a shared session key.

In order to have properly secure communications, the cryptographically strong pseudo-random number generator (PRNG) in TNL must be initialized with good (high entropy) random data.

## Useful Classes

TNL uses a number of utility classes throughout. Some of these include:

**Vector** – The Vector template class is a lightweight version of the STL Vector container.

**DataChunker** – The DataChunker class performs small, very low overhead memory allocation out of a pool. Individual allocations cannot be freed, however the entire pool can be freed at once.

**ClassChunker** – The ClassChunker template class uses the DataChunker to manage allocation and deallocation of instances of the specific class. ClassChunker maintains a free list for quick allocation and deallocation.

**StringTableEntry** – StringTableEntry wraps a ref-counted element in the StringTable, with simple conversion operations to C strings. StringTableEntry instances can be sent in the parameter lists of RPC methods or using the EventConnection::packStringTableEntry method.

**ByteBuffer** – The ByteBuffer class wraps a buffer of arbitrary binary data. ByteBuffer serves as a base class for BitStream, as well as the cryptographic primitives.

## Debugging and Error Handling

Correcting bugs in a networked simulation can sometimes be an incredibly difficult task. TNL provides some handy features to make debugging somewhat less challenging:

**Asserts** - The TNLAssert and TNLAssertV macros specify a condition that, if false, will cause the application to display an error dialog and halt in the debugger where the assert was hit. Asserts are very useful for sanity-checking arguments to functions, and making sure network data is being properly read from and written to packets.

**Logging** - TNL has a set of simple but effective facilities for logging status information. The TNLLogMessage and TNLLogMessageV macros allow the user to specify logging tokens with particular log messages. Logging of messages associated with a given token can be enabled or disabled using the TNLLogEnable macro. Also, the TNLLogBlock macro can be used to log multiple logprintf statements in a single test. Rather than log to a file or other destination, TNL applications must declare at least one instance of a subclass of TNLLogConsumer. Every TNLLogConsumer instance receives all the logging messages that are currently enabled.

**Debugging object writes into packets** - One of the most common errors programmers experience when using TNL is failing to properly match the NetEvent::pack and NetEvent::unpack or NetObject::packUpdate and NetObject::unpackUpdate routines. By default, if TNL\_DEBUG is defined, the EventConnection and GhostConnection classes will write extra size information into the packet before each object's data are written. The ConnectionParameters::mDebugObjectSizes flag can be modified directly to further control this behavior.

## Remote Procedure Calls (RPC)

The Torque Network Library has a powerful, yet simple to use Remote Procedure Call framework for passing information through a network connection. Subclasses of EventConnection and NetObject can declare member functions using the RPC macros so that when called the function arguments are sent to the remote EventConnection or NetObject(s) associated with that object.

For example, suppose you have a connection class called SimpleEventConnection:

```
class SimpleEventConnection : public EventConnection {
```



```

public:
    TNL_DECLARE_RPC(rpcPrintString, (StringPtr theString, U32 messageCount));
};

TNL_IMPLEMENT_RPC(SimpleEventConnection, rpcPrintString,
    (StringPtr theString, messageCount), (theString, messageCount),
    NetClassGroupGameMask, RPCGuaranteedOrdered, RPCDirAny, 0) {

    for(U32 i = 0; i < messageCount; i++)
        printf("%s", theString.getString());
}

...
void somefunction(SimpleEventConnection *connection)
{
    connection->rpcPrintString("Hello World!", 5);
}

```

In this example the class `SimpleEventConnection` is declared to have a single RPC method named `rpcPrintString`. The `TNL_DECLARE_RPC` macro can just be viewed as a different way of declaring a class's member functions, with the name as the first argument and the parenthesized parameter list as the second. Since RPC calls execute on a remote host, they never have a return value - although a second RPC could be declared to pass messages in the other direction.

The body of the RPC method is declared using the `TNL_IMPLEMENT_RPC` macro, which has some additional arguments: the named parameter list without the types, which `NetClassMask` the RPC is valid in, what level of data guarantee it uses, the direction it is allowed to be called on the connection and a version number. The body of the function, which in this case prints the passed message "Hello, World!" 5 times to stdout, is executed on the remote host from which the method was originally invoked.

As the `somefunction` code demonstrates, RPC's are invoked in the same way as any other member function in the class.

RPCs behave like virtual functions in that their bodies can be overridden in subclasses that want to implement new behaviour for the message. The class declaration for an overridden RPC should include the `TNL_DECLARE_RPC_OVERRIDE` macro used for each method that will be redefined. The `TNL_IMPLEMENT_RPC_OVERRIDE` macro should be used outside the declaration of the class to implement the body of the new RPC.

Internally the RPC macros construct new `NetEvent` classes and encapsulate the function call arguments using the `FunctorDecl` template classes. By default the following types are allowed as parameters to RPC methods:

- S8, U8
- S16, U16
- S32, U32
- F32
- Int<>
- SignedInt<>
- Float<>
- SignedFloat<>
- RangedU32<>
- bool
- StringPtr
- StringTableEntry
- ByteBufferPtr
- IPAddress
- Vector<> of all the preceding types

New types can be supported by implementing a template override for the `Types::read` and `Types::write` functions. All arguments to RPCs must be passed by value (ie no reference or pointer types).

The `Int`, `SignedInt`, `Float`, `SignedFloat` and `RangedU32` template types use the template parameter(s) to specify the number of bits necessary to transmit that variable across the network. For example:

```

...
TNL_DECLARE_RPC(someTestFunction, (Int<4> fourBitInt, SignedFloat<7> aFloat,
    RangedU32<100, 199> aRangedU32);
...

```

The preceding RPC method would use  $4 + 7 + 7 = 18$  bits to transmit the arguments to the function over the network, not including the RPC event overhead.

## 3D Engine OGRE

[Information extracted from [www.ogre3d.com](http://www.ogre3d.com)]

### Features

#### Productivity features

- Simple, easy to use OO interface designed to minimise the effort required to render 3D scenes, and to be independent of 3D implementation i.e. Direct3D/OpenGL.
- Extensible example framework makes getting your application running is quick and simple
- Common requirements like render state management, hierarchical culling, dealing with transparency are done for you automatically saving you valuable time
- Clean, uncluttered design and full documentation of all engine classes

#### Platform & 3D API support

- Direct3D and OpenGL support
- Windows (all major versions), Linux and Mac OSX support
- Builds on Visual C++ 6 (with STLport), Visual C++.Net 2002 (with STLport), Visual C++.Net 2003 on Windows
- Builds on gcc 3+ on Linux / Mac OSX (using Xcode)

#### Material / Shader support

- Powerful material declaration language allows you to maintain material assets outside of your code
- Supports vertex and fragment programs (shaders), both low-level programs written in assembler, and high-level programs written in Cg, DirectX9 HLSL, or GLSL and provides automatic support for many commonly bound constant parameters like worldview matrices, light state information, object space eye position etc
- Supports the complete range of fixed function operations such as multitexture and multipass blending, texture coordinate generation and modification, independent colour and alpha operations for non-programmable hardware or for lower cost materials
- Multiple pass effects, with pass iteration if required for the closest 'n' lights
- Support for multiple material techniques means you can design in alternative effects for a wide range of cards and OGRE automatically uses the best one supported
- Material LOD support; your materials can reduce in cost as the objects using them get further away
- Load textures from PNG, JPEG, TGA, BMP or DDS files, including unusual formats like 1D textures, volumetric textures, cubemaps and compressed textures (DXT/S3TC)
- Textures can be provided and updated in realtime by plugins, for example a video feed
- Easy to use projective texturing support

#### Meshes

- Flexible mesh data formats accepted, separation of the concepts of vertex buffers, index buffers, vertex declarations and buffer mappings

- Export from many modelling tools including Milkshape3D, 3D Studio Max, Maya, Blender and Wings3D
- Skeletal animation, including blending of multiple animations, variable bone weight skinning, and hardware-accelerated skinning
- Biquadric Bezier patches for curved surfaces
- Progressive meshes (LOD)
- Static geometry batcher

### Scene Features

- Highly customisable, flexible scene management, not tied to any single scene type. Use predefined classes for scene organisation if they suit or plug in your own subclass to gain full control over the scene organisation
- Several example plugins demonstrate various ways of handling the scene specific to a particular type of layout (e.g. BSP, Octree)
- Hierarchical scene graph; nodes allow objects to be attached to each other and follow each others movements, articulated structures etc
- Multiple shadow rendering techniques, each highly configurable and taking full advantage of any hardware acceleration available.
- Scene querying features

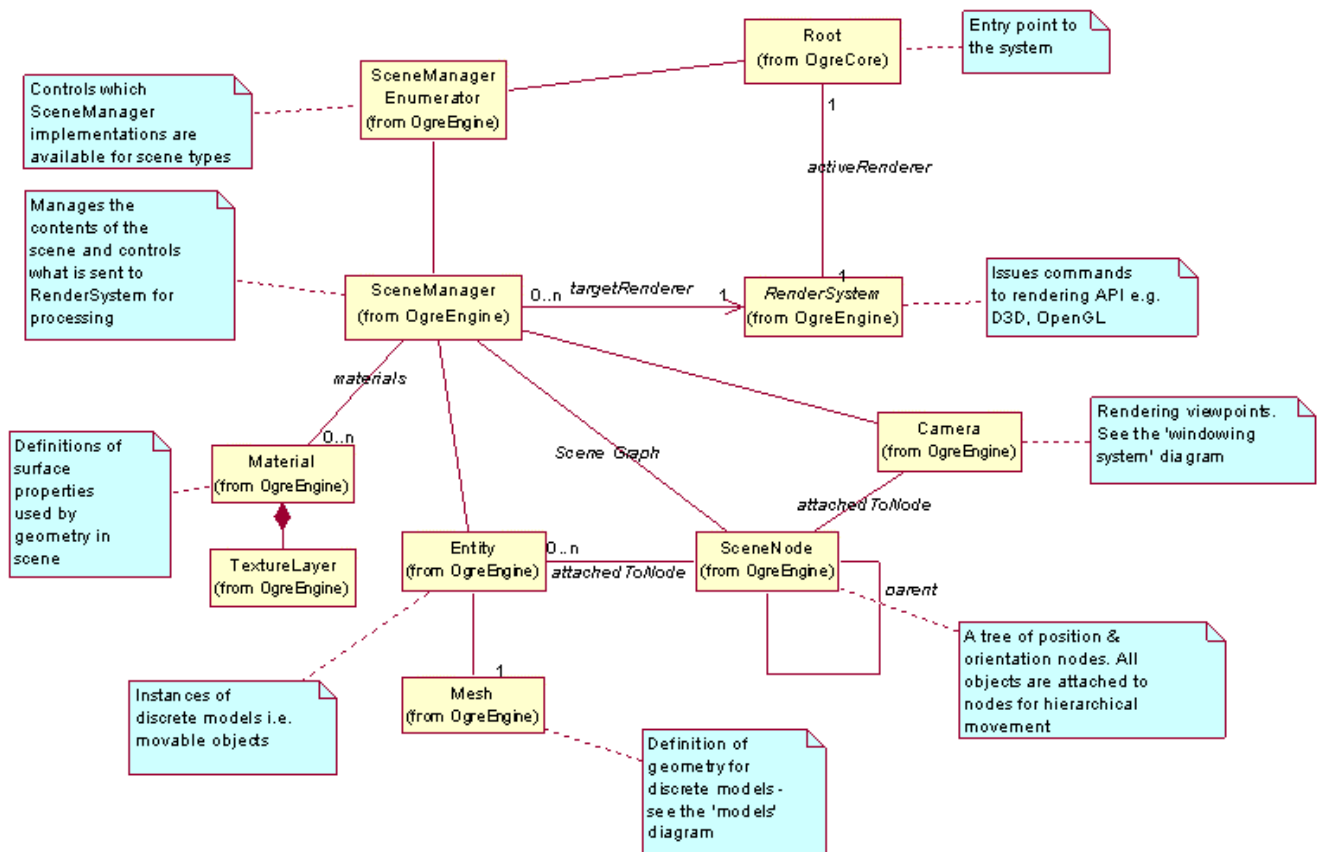
### Special Effects

- Particle Systems, including easily extensible emitters, affectors and renderers (customisable through plugins). Systems can be defined in text scripts for easy tweaking. Automatic use of particle pooling for maximum performance
- Support for skyboxes, skyplanes and skydomes, very easy to use
- Billboarding for sprite graphics
- Transparent objects automatically managed (rendering order & depth buffer settings all set up for you)

### Misc features

- Common resource infrastructure for memory management and loading from archives (ZIP, PK3)
- Flexible plugin architecture allows engine to be extended without recompilation
- 'Controllers' allow you to easily organise derived values between objects e.g. changing the colour of a ship based on shields left
- Debugging memory manager for identifying memory leaks
- ReferenceAppLayer provides an example of how to combine OGRE with other libraries, for example ODE for collision & physics
- XMLConverter to convert efficient runtime binary formats to/from XML for interchange or editing

## UML Overview



## SceneManagers

Everything that appears on the screen is managed by the SceneManager. When you place objects in the scene, the SceneManager is the class which keeps track of their locations. When you create Cameras to view the scene the SceneManager keeps track of them. When you create planes, billboards, lights... the SceneManager keeps track of them.

### Selecting a Scene Manager

There are multiple types of SceneManagers. You can select one via the *getSceneManager* method defined in your root node:

```
mRoot->getSceneManager (ST_GENERIC);
```

The argument to *getSceneManager* specifies the type of scene manager to use, based on the following values:

- ST\_GENERIC - Generic scene manager (Octree if you load Plugin\_OctreeSceneManager, DotScene if you load Plugin\_DotSceneManager)
- ST\_EXTERIOR\_CLOSE - Terrain\_Scene\_Manager
- ST\_EXTERIOR\_FAR - Nature scene manager
- ST\_EXTERIOR\_REAL\_FAR - Paging\_Scene\_Manager
- ST\_INTERIOR - BSP scene manager

### Octree Scene Manager

Uses an octree to split the scene and performs well for most scenes, except those which are reliant on heavy occlusion.

**Pros:**

- A simple and generic solution, works well for most scenes
- Can use the StaticGeometry class to accelerate large chunks of immovable geometry

**Cons:**

- No specific acceleration for particular scene structures
- Heavily occluded scenes will need a more specialised solution

***Terrain Scene Manager***

The terrain\_scene\_manager is intended for relatively small scenes involving static terrain. This scene manager makes it easy to generate terrain from a heightmap. Heightmaps can be created from a variety of tools.

**Pros:**

- Fast rendering of high-resolution terrain (due to efficient level of detail and culling algorithms)
- Easy to generate terrain via heightmaps and terrain textures
- Vertex program based morphing between LOD levels
- Customisable material so you can use shaders if you want

**Cons:**

- No paging out of the box - the interface hooks are there but you need to add it
- No splatting out of the box, but can be done with a custom material

***Nature Scene Manager (ogreaddons)***

The nature scene manager is intended for larger outdoor scenes than those provided by the terrain scene manager. The same heightmap is repeated over a set of terrain tiles, each of which can use a different terrain texture.

**Pros:**

- Handles larger scenes than the terrain scene manager

**Cons:**

- Terrains have substantially less detail than with the terrain scene manager

***Paging Scene Manager (ogreaddons)***

The Paging\_Scene\_Manager allows scenes to be split into a set of *pages*. Only those pages that are being used need be loaded at any given time, allowing arbitrarily large scenes. Each page has its own heightmap, to which several textures can be applied by height. (This allows snowy-peaked mountains within a green landscape, for example.)

**Pros:**

- Handles larger scenes than both the terrain and nature scene managers
- Allows Real-time Terrain deformation and saves to files.
- Allows multiple heightmaps and multiple textures per heightmap:
  - Image + detail (fading on distance)
  - Texture coloring based on user height (4 colors)
  - Splatting.
  - Real-time Splatting using Shaders.
  - Real-time Texture coloring based on user height (4 colors)
- Special Video Card Memory Savings that divides at least per 3 Memory consumption:
  - Texture Coordinates sharing accross pages (one texture coordinates set for any number of pages.)
  - Displacement Mapping using shaders (use only one float for a vertices instead of 3.)
- Map Tool (called "Mapsplitter") that split Big Map and Textures in pages, can compute lightmaps, normal maps, splatting maps.

- Support Raw Heightmap up to 16 bits per height (instead of 8 bits jpg,png files)
- Real-time Map change, Texturing Change
- Binary demo: <http://tuan.kuran.es.free.fr/Ogre.html>
- Vertex Morphing using shaders
- Horizon Occlusion Visibility Real-time determination : everything behind a mountains will not be send to video card.
- Octree support

**Cons:**

- Requires installation of the paging scene manager plug-in
- Needs use of The Map Tool to generates pages
- More options means more complexity.

***BSP Scene Manager***

This scene manager is intended for use in interior scenes. Particularly, it is optimized for the sort of geometry that results from intersecting walls and corridors.

**Pros:**

- Optimized for interior scenes

***Dot Scene Manager (ogreaddons)***

The Dot\_Scene\_Manager allows the geometry and meshes of a scene to be stored in a single file.

**Pros:**

- Contains scene in a single file
- Allows meshes to be classified as static or dynamic
- Octree support

**Cons:**

- Requires installation of the dot scene manager plug-in
- Needs use of a processing tool to build a .scene file

**Entities**

An Entity is one of the types of object that you can render on a scene. You can think of an Entity as being anything that's represented by a 3D mesh. A robot would be an entity, a fish would be an entity, the terrain your characters walk on would be a very large entity. Things such as Lights, Billboards, Particles, Cameras, etc would not be an Entity.

One thing to note about Ogre is that it separates renderable objects from their location and orientation. This means that you cannot directly place an Entity in a scene. Instead you must attach the Entity to a SceneNode object, and this SceneNode contains the information about location and orientation.

**SceneNodes**

As already mentioned, SceneNodes keep track of location and orientation for all of the objects attached to it. When you create an Entity, it is not ever rendered on the scene until you attach it to a SceneNode. Similarly, a SceneNode is not an object that is displayed on the screen. Only when you create a SceneNode and attach an Entity (or other object) to it is something actually displayed on the screen.

SceneNodes can have any number of objects attached to them. Lets say you have a character walking around on the screen and you want to have him generate a light around him. The way you do this would be to first create a SceneNode, then create an Entity for the character and attach it to the SceneNode. Then you would create a Light object and attach it to the SceneNode. SceneNodes may also be attached to other SceneNodes which allows you to create entire hierarchies of nodes.

One major concept to note about SceneNodes is that a SceneNode's position is **always** relative to its parent SceneNode, and each SceneManager contains a root node which all other SceneNodes are attached.

## Coordinates and Vectors

Ogre (like many graphics engines) uses the x and z axis as the horizontal plane, and the y axis as your vertical axis. As you are looking at your monitor now, the x axis would run from the left side to the right side of your monitor, with the right side being the positive x direction. The y axis would run from the bottom of your monitor to the top of your monitor, with the top being the positive y direction. The z axis would run into and out of your screen, with out of the screen being the positive z direction.

Ogre uses the Vector class to represent both position **and** direction (there is no Point class). There are vectors defined for 2 (Vector2), 3 (Vector3), and 4 (Vector4) dimensions, with Vector3 being the most commonly used.

### Vector

**[Wikipedia<sup>10</sup> definition]** Informally, a **vector** is a quantity characterized by a number (indicating magnitude) and a direction, often represented graphically by an arrow. Examples are "moving north at 90 km/h" or "pulling towards the center of Earth with a force of 70 newtons".

The notion of having a "magnitude" and "direction" is formalized by saying that the vector has components that transform like the coordinates under rotations. That is, if the coordinate system undergoes a rotation described by a rotation matrix  $R$ , so that a coordinate vector  $\mathbf{x}$  is transformed to  $\mathbf{x}' = R\mathbf{x}$ , then any other vector  $\mathbf{v}$  is similarly transformed via  $\mathbf{v}' = R\mathbf{v}$ .

More generally, a vector is a tensor of contravariant rank one. In differential geometry, the term *vector* usually refers to quantities that are closely related to tangent spaces of a differentiable manifold (assumed to be three-dimensional and equipped with a positive definite Riemannian metric). (A four-vector is a related concept when dealing with a 4 dimensional spacetime manifold in relativity.)

Examples of vectors include displacement, velocity, electric field, momentum, force and acceleration.

Vectors can be contrasted with scalar quantities such as distance, speed, energy, time, temperature, charge, power, work and mass, which have magnitude, but no direction (they are invariant under coordinate rotations). The magnitude of any vector is a scalar.

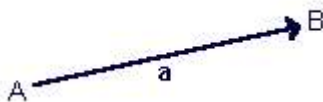
A related concept is that of a pseudovector (or **axial vector**). This is a quantity that transforms like a vector under proper rotations, but gains an additional sign flip under improper rotations. Examples of pseudovectors include magnetic field, torque and angular momentum. (This distinction between vectors and pseudovectors is often ignored, but it becomes important in studying symmetry properties.) To distinguish from pseudo/axial vectors, an ordinary vector is sometimes called a **polar vector**.

Sometimes, one speaks informally of *bound* or *fixed* vectors, which are vectors additionally characterized by a "base point". Most often, this term is used for position vectors (relative to an origin point). More generally, however, the physical interpretation of a particular vector can be parameterized by any number of quantities.

### Representation of a vector

Symbols standing for vectors are usually printed in boldface as  $\mathbf{a}$ ; this is also the convention adopted in this encyclopedia. Other conventions includes  $\underline{a}$ , especially in handwriting.  $\vec{a}$  Alternately, some use a tilde ( $\sim$ ) placed under the vector. The length or magnitude or norm of the vector  $\mathbf{a}$  is denoted by  $|\mathbf{a}|$ .

Vectors are usually shown in graphs or other diagrams as arrows, as illustrated below:



Here the point  $A$  is called the *tail*, *base*, *start*, or *origin*; point  $B$  is called the *head*, *tip*, *endpoint*, or *destination*. The length of the

<sup>10</sup> Free online encyclopedia: <http://en.wikipedia.org/>

arrow represents the vector's magnitude, while the direction in which the arrow points represents the vector's direction.

In the figure above, the arrow can also be written as  $\underline{AB}$

In order to calculate with vectors, the graphical representation is too cumbersome. Vectors in a  $n$ -dimensional Euclidean space can be represented as a linear combination of  $n$  mutually perpendicular *unit vectors*. In this article, we will consider  $\mathbf{R}^3$  as an example. In  $\mathbf{R}^3$ , we usually denote the unit vectors parallel to the  $x$ -,  $y$ - and  $z$ -axes by  $\mathbf{i}$ ,  $\mathbf{j}$  and  $\mathbf{k}$  respectively. Any vector  $\mathbf{a}$  in  $\mathbf{R}^3$  can be written as  $\mathbf{a} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$  with real numbers  $a_1$ ,  $a_2$  and  $a_3$  which are uniquely determined by  $\mathbf{a}$ . Sometimes  $\mathbf{a}$  is then also written as a 3-by-1 or 1-by-3 matrix:

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$$\mathbf{a} = (a_1 \quad a_2 \quad a_3)$$

even though this notation suppresses the dependence of the coordinates  $a_1$ ,  $a_2$  and  $a_3$  on the specific choice of coordinate system  $\mathbf{i}$ ,  $\mathbf{j}$  and  $\mathbf{k}$ .

## Length of a vector

The length of the vector  $\mathbf{a} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$  can be computed as

$$|\mathbf{a}| = \sqrt{a_1^2 + a_2^2 + a_3^2}$$

which is a consequence of the Pythagorean theorem: The sum of the areas of the squares on the legs of a right triangle is equal to the area of the square on the hypotenuse.

## Vector equality

Two vectors are said to be equal if they have the same magnitude and direction. However if we are talking about bound vector, then two bound vectors are equal if they have the same base point and end point.

For example, the vector  $\mathbf{i} + 2\mathbf{j} + 3\mathbf{k}$  with base point (1,0,0) and the vector  $\mathbf{i} + 2\mathbf{j} + 3\mathbf{k}$  with base point (0,1,0) are different bound vectors, but the same (unbounded) vector.

## Vector addition and subtraction

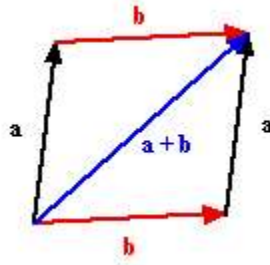
Let  $\mathbf{a} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$  and  $\mathbf{b} = b_1\mathbf{i} + b_2\mathbf{j} + b_3\mathbf{k}$ .

The sum of  $\mathbf{a}$  and  $\mathbf{b}$  is:

$$\mathbf{a} + \mathbf{b} = (a_1 + b_1)\mathbf{i} + (a_2 + b_2)\mathbf{j} + (a_3 + b_3)\mathbf{k}$$

The addition may be represented graphically by placing the start of the arrow  $\mathbf{b}$  at the tip of the arrow  $\mathbf{a}$ , and then drawing an arrow from the start of  $\mathbf{a}$  to the tip of  $\mathbf{b}$ . The new arrow drawn represents the vector  $\mathbf{a} + \mathbf{b}$ , as illustrated below:



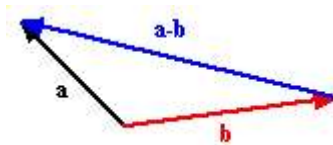


This addition method is sometimes called the *parallelogram rule* because  $\mathbf{a}$  and  $\mathbf{b}$  form the sides of a parallelogram and  $\mathbf{a} + \mathbf{b}$  is one of the diagonals. If  $\mathbf{a}$  and  $\mathbf{b}$  are bound vectors, then the addition is only defined if  $\mathbf{a}$  and  $\mathbf{b}$  have the same base point, which will then also be the base point of  $\mathbf{a} + \mathbf{b}$ . One can check geometrically that  $\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$  and  $(\mathbf{a} + \mathbf{b}) + \mathbf{c} = \mathbf{a} + (\mathbf{b} + \mathbf{c})$ .

The difference of  $\mathbf{a}$  and  $\mathbf{b}$  is:

$$\mathbf{a} - \mathbf{b} = (a_1 - b_1)\mathbf{i} + (a_2 - b_2)\mathbf{j} + (a_3 - b_3)\mathbf{k}$$

Subtraction of two vectors can be geometrically defined as follows: to subtract  $\mathbf{b}$  from  $\mathbf{a}$ , place the ends of  $\mathbf{a}$  and  $\mathbf{b}$  at the same point, and then draw an arrow from the tip of  $\mathbf{b}$  to the tip of  $\mathbf{a}$ . That arrow represents the vector  $\mathbf{a} - \mathbf{b}$ , as illustrated below:



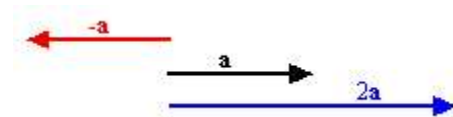
If  $\mathbf{a}$  and  $\mathbf{b}$  are bound vectors, then the subtraction is only defined if they share the same base point which will then also become the base point of their difference. This operation deserves the name "subtraction" because  $(\mathbf{a} - \mathbf{b}) + \mathbf{b} = \mathbf{a}$ .

## Scalar multiplication

A vector may also be multiplied by a real number  $r$ . Numbers are often called **scalars** to distinguish them from vectors, and this operation is therefore called **scalar multiplication**. The resulting vector is:

$$r\mathbf{a} = (ra_1)\mathbf{i} + (ra_2)\mathbf{j} + (ra_3)\mathbf{k}$$

The length of  $r\mathbf{a}$  is  $|r||\mathbf{a}|$ . If the scalar is negative, it also changes the direction of the vector by  $180^\circ$ . Two examples ( $r = -1$  and  $r = 2$ ) are given below:

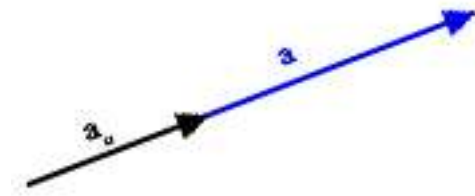


Here it is important to check that the scalar multiplication is compatible with vector addition in the following sense:  $r(\mathbf{a} + \mathbf{b}) = r\mathbf{a} + r\mathbf{b}$  for all vectors  $\mathbf{a}$  and  $\mathbf{b}$  and all scalars  $r$ . One can also show that  $\mathbf{a} - \mathbf{b} = \mathbf{a} + (-1)\mathbf{b}$ .

The set of all geometrical vectors, together with the operations of vector addition and scalar multiplication, satisfies all the axioms of a vector space. Similarly, the set of all bound vectors with a common base point forms a vector space. This is where the term "vector space" originated.

## Unit vector

A unit vector is any vector with a length of one. If you have a vector of arbitrary length, you can use it to create a unit vector. This is known as **normalizing** a vector.



To normalize a vector, scale the vector by the inverse of its length. That is:

$$\mathbf{a}_u = \frac{\mathbf{a}}{|\mathbf{a}|} = \frac{a_1}{|\mathbf{a}|} \mathbf{i} + \frac{a_2}{|\mathbf{a}|} \mathbf{j} + \frac{a_3}{|\mathbf{a}|} \mathbf{k}$$

## Dot product

The dot product of two vectors  $\mathbf{a}$  and  $\mathbf{b}$  (also called the *inner product*, or, since its result is a scalar, the *scalar product*) is denoted by  $\mathbf{a} \cdot \mathbf{b}$  or sometimes by  $(\mathbf{a}, \mathbf{b})$  and is defined as:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos(\theta)$$

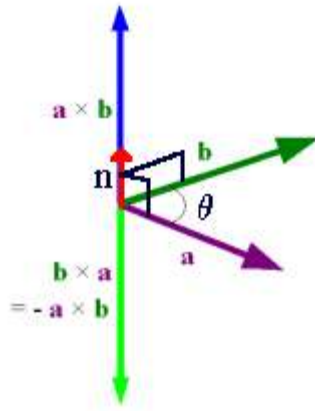
where  $\theta$  is the measure of the angle between  $\mathbf{a}$  and  $\mathbf{b}$  (see trigonometric function for an explanation of cosine). Geometrically, this means that  $\mathbf{a}$  and  $\mathbf{b}$  are drawn with a common start point and then the length of  $\mathbf{a}$  is multiplied with the length of that component of  $\mathbf{b}$  that points in the same direction as  $\mathbf{a}$ . This operation is often useful in physics; for instance, work is the dot product of force and displacement.

## Cross product

The cross product (also *vector product* or *outer product*) differs from the dot product primarily in that the result of a cross product of two vectors is a vector. While everything that was said above can be generalized in a straightforward manner to more than three dimensions, the cross product is only meaningful in three dimensions (although a related product exists in seven dimensions - see below). The cross product, denoted  $\mathbf{a} \times \mathbf{b}$ , is a vector perpendicular to both  $\mathbf{a}$  and  $\mathbf{b}$  and is defined as:

$$\mathbf{a} \times \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \sin(\theta) \mathbf{n}$$

where  $\theta$  is the measure of the angle between  $\mathbf{a}$  and  $\mathbf{b}$ , and  $\mathbf{n}$  is a unit vector perpendicular to both  $\mathbf{a}$  and  $\mathbf{b}$ . The problem with this definition is that there are *two* unit vectors perpendicular to both  $\mathbf{b}$  and  $\mathbf{a}$ . Which vector is the correct one depends upon the *orientation* of the vector space, i.e. on the *handedness* of the coordinate system. The coordinate system  $\mathbf{i}, \mathbf{j}, \mathbf{k}$  is called *right handed*, if the three vectors are situated like the thumb, index finger and middle finger (pointing straight up from your palm) of your right hand. Graphically the cross product can be represented by this figure



In such a system,  $\mathbf{a} \times \mathbf{b}$  is defined so that  $\mathbf{a}$ ,  $\mathbf{b}$  and  $\mathbf{a} \times \mathbf{b}$  also becomes a right handed system. If  $\mathbf{i}$ ,  $\mathbf{j}$ ,  $\mathbf{k}$  is left-handed, then  $\mathbf{a}$ ,  $\mathbf{b}$  and  $\mathbf{a} \times \mathbf{b}$  is defined to be left-handed. Because the cross product depends on the choice of coordinate systems, its result is referred to as a pseudovector. Fortunately, in nature cross products tend to come in pairs, so that the "handedness" of the coordinate system is undone by a second cross product.

The length of  $\mathbf{a} \times \mathbf{b}$  can be interpreted as the area of the parallelogram having  $\mathbf{a}$  and  $\mathbf{b}$  as sides.

## Scalar triple product

The *scalar triple product* (also called the *box product* or *mixed triple product*) isn't really a new operator, but a way of applying the other two multiplication operators to three vectors. The scalar triple product is denoted by  $(\mathbf{a} \ \mathbf{b} \ \mathbf{c})$  and defined as:

$$(\mathbf{a} \ \mathbf{b} \ \mathbf{c}) = \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$$

It has three primary uses. First, the absolute value of the box product is the volume of the parallelepiped which has edges that are defined by the three vectors. Second, the scalar triple product is zero if and only if the three vectors are linearly dependent, which can be easily proved by considering that in order for the three vectors to not make a volume, they must all lie in the same plane. Third, the box product is positive if and only if the three vectors  $\mathbf{a}$ ,  $\mathbf{b}$  and  $\mathbf{c}$  are oriented like the coordinate system  $\mathbf{i}$ ,  $\mathbf{j}$  and  $\mathbf{k}$ .

In coordinates, if the three vectors are thought of as rows, the scalar triple product is simply the determinant of the 3-by-3 matrix having the three vectors as rows. The scalar triple product is linear in all three entries and anti-symmetric in the following sense:

$$(\mathbf{a} \ \mathbf{b} \ \mathbf{c}) = (\mathbf{c} \ \mathbf{a} \ \mathbf{b}) = (\mathbf{b} \ \mathbf{c} \ \mathbf{a}) = -(\mathbf{a} \ \mathbf{c} \ \mathbf{b}) = -(\mathbf{b} \ \mathbf{a} \ \mathbf{c}) = -(\mathbf{c} \ \mathbf{b} \ \mathbf{a})$$

Technically, the scalar triple product isn't a scalar, it is a *pseudoscalar*: under a coordinate inversion ( $\mathbf{x}$  goes to  $-\mathbf{x}$ ), it flips sign.

## Cameras

A Camera is what we use to view the scene that we have created. A Camera is a special object which works somewhat like a SceneNode does. The Camera object has setPosition, yaw, roll and pitch functions, and you can attach it to any SceneNode. Just like SceneNodes, a Camera's position is relative to its parents

## Viewports

The concept of a Viewport class will become much more useful if multiple cameras are used. It is important to understand how Ogre decides which Camera to render a scene. It is possible in Ogre to have multiple SceneManagers running at the same time. It is

also possible to split the screen up into multiple areas, and have separate cameras render to separate areas on the screen (example: a split view for 2 players in a console game).

To understand how Ogre renders a scene, we have to consider three of Ogre's constructs: the Camera, the SceneManager, and the RenderWindow. The RenderWindow is basically the window in which everything is displayed. The SceneManager object creates Cameras to view the scene. It must be told to the RenderWindow which Cameras to display on the screen, and what portion of the window to render it in. The area in which is told to the RenderWindow to display the Camera is the Viewport. Under most typical uses of Ogre, it will generally create only one Camera, register the Camera to use the entire RenderWindow, and thus only have one Viewport object.

## Shadows

Ogre currently supports three types of Shadows:

1. Modulative Texture Shadows (SHADOWTYPE\_TEXTURE\_MODULATIVE) - The least intensive of the three. This creates a black and white render-to-texture of shadow casters, which is then applied to the scene.
2. Modulative Stencil Shadows (SHADOWTYPE\_STENCIL\_MODULATIVE) - This technique renders all shadow volumes as a modulation after all non-transparent objects have been rendered to the scene. This is not as intensive as Additive Stencil Shadows, but it is also not as accurate.
3. Additive Stencil Shadows (SHADOWTYPE\_STENCIL\_ADDITIVE) - This technique renders each each light as a separate additive pass on the scene. This is very hard on the graphics card because each additional light requires an additional pass at rendering the scene.

## Lights

There are three types of lighting that Ogre provides.

1. Point (LT\_POINT) - Point light sources emit light from them in every direction.
2. Spotlight (LT\_SPOTLIGHT) - A spotlight works exactly like a flashlight does. You have a position where the light starts, and then light heads out in a direction. You can also tell the light how large of an angle to use for the inner circle of light and the outer circle of light (you know how flashlights are brighter in the center, then lighter after a certain point?).
3. Directional (LT\_DIRECTIONAL) - Directional light simulates far away light that hits everything in the scene from a direction. Lets say you have a night time scene and you want to simulate moonlight. You could do this by setting the ambient light for the scene, but that's not exactly realistic since the moon does not light everything equally (neither does the sun). One way to do this would be to set a directional light and point in the direction the moon would be shining.

## Sky

Ogre prodvides three different types of sky: SkyBoxes, SkyDomes, and SkyPlanes.

### SkyBoxes

A SkyBox is basically a giant cube that surrounds all of the objects in the scene.

### SkyDomes

SkyDomes are very similar to SkyBoxes, a giant cube is created around the Camera and rendered onto, but the biggest difference is the texture is "projected" onto the SkyBox in a spherical manner. You are still looking at a cube, but it looks as if the texture is wrapped around the surface of a sphere. The primary drawback to this method is that the bottom of the cube will be untextured, so you always need to have some type of terrain that hides the base.

### SkyPlanes

SkyPlanes are very different from SkyBoxes and SkyDomes. Instead of a cube to render the sky texture on, we use just a single plane.

## ***Which to Use?***

Which sky to use depends entirely on the application. If you have to see all around you, even in the negative y direction, then really your only real choice is to use a SkyBox. If you have terrain, or some kind of floor which blocks the view of the negative y direction, then using a SkyDome seems to give more realistic results. For areas where you cannot see to the horizon (such as a valley surrounded by mountains on all sides, or the inner courtyard of a castle), a SkyPlane will give you very good looking results for very little GPU costs. The primary reason to use a SkyPlane is because it plays nicely with fog effects.

## **FrameListeners**

Ogre's main loop (Root::startRendering):

1. The Root object calls the frameStarted method on all registered FrameListeners.
2. The Root object renders one frame.
3. The Root object calls the frameEnded method on all registered FrameListeners.

To control different inputs it is possible to merge the FrameListener with a KeyListener, MouseListener and MouseMotionListener.

## ***KeyListener***

The KeyListener interface defines several functions for getting keyboard input. The keyPressed method is called when a key is pressed (that is, goes down). The keyReleased function is called when the key comes back up. The keyClicked method is called when a key is pressed and then comes back up (which is almost always equivalent to keyReleased). Whenever a key is pressed, three events are generated: one when the key is pressed down (keyPressed), one when the key is let up (keyReleased), and one after the previous two (keyClicked).

## ***MouseListener***

The MouseListener interface defines functions for getting mouse click input. The mousePressed function is called when the user depresses a mouse button. The mouseReleased function is called when the user releases a mouse button. The mouseClicked function is broken and does not work, so do not try to use it (but since it is pure virtual you always have to define it). The mouseEntered and mouseExited functions are both legacy code which we will not use, but since they are pure virtual we have to define an empty function for them anyway.

## ***MouseMotionListener***

The MouseMotionListener interface defines functions for getting mouse movement events. The mouseMoved function is called whenever the mouse is moved, and the mouseDragged function is called whenever the mouse is moved when a button is held down. The mouseDragMoved function seems to be broken (at least I cannot find anything that makes it actually get called).

# **Implementation**

## ***Requirements***

### **GNU/Linux**

It is needed:

- DevIL 1.6.5 (newer version does not work properly with CEGUI). It is possible to find precompiled packages for Ubuntu: <http://ftp.ankara.edu.tr/ubuntu/pool/universe/d/devil/>
- CEGUI 0.2.0
- OGRE 1.0.1
- OpenTNL 1.5.0

When everything is compiled, we must change the Makefile's of the project in order to indicate where is the source code available. Variables:

```
ogre_src_dir = /home/marble/novuz-files/ogrenew  
tnl_src_dir = /home/marble/novuz-files/tnl
```

To compile the project in the “demo” directory:

```
cd network  
make  
cd ../interface  
make
```

So we get the following executables:

```
network/GameServer  
network/gameClient
```

First we must run the server:

```
cd network  
./GameServer localhost:5555
```

And then connect our client (it is possible to execute several clients):

```
cd interface  
./gameClient
```

A OpenTNL test is also included in the Linux version, it is called “tnl.scope” and it demonstrates how OpenTNL can work with scopes. In order to compile it, we just need to change the Makefile's variable pointing to the TNL source code:

```
tnl_src_dir = /home/marble/novuz-files/tnl
```

And execute:

```
make
```

Now we can start the server and as many clients as we want:

```
./GameServer localhost:5555  
./VisualGameClient localhost:5555
```

## MS Windows

We need:

- Visual Studio .NET 2003
- OGRE SDK 1.0.1 compiled for Visual Studio .NET 2003
- OpenTNL 1.5.0 (can be compiled from Visual Studio .NET 2003)

Now we have to open the VS.NET solution and edit the different project's properties in order to match them with the good location of OpenTNL and OGRE.

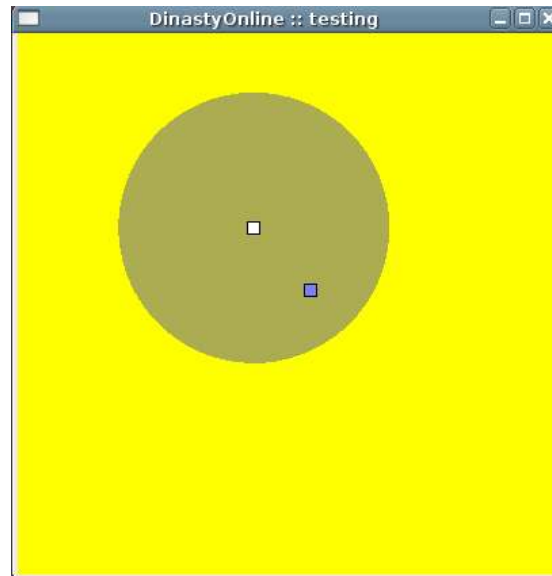
**NOTE:** In Windows there are problems with the OGRE Memory Manager and OpenTNL.

## Screenshots

Test OpenTNL:

- White square: our player
- Green circle: our scope
- Blue square: another player that is in our scope, every out our scope is not showed.

We can move our player clicking in any place of the window.



Demo, login screen:

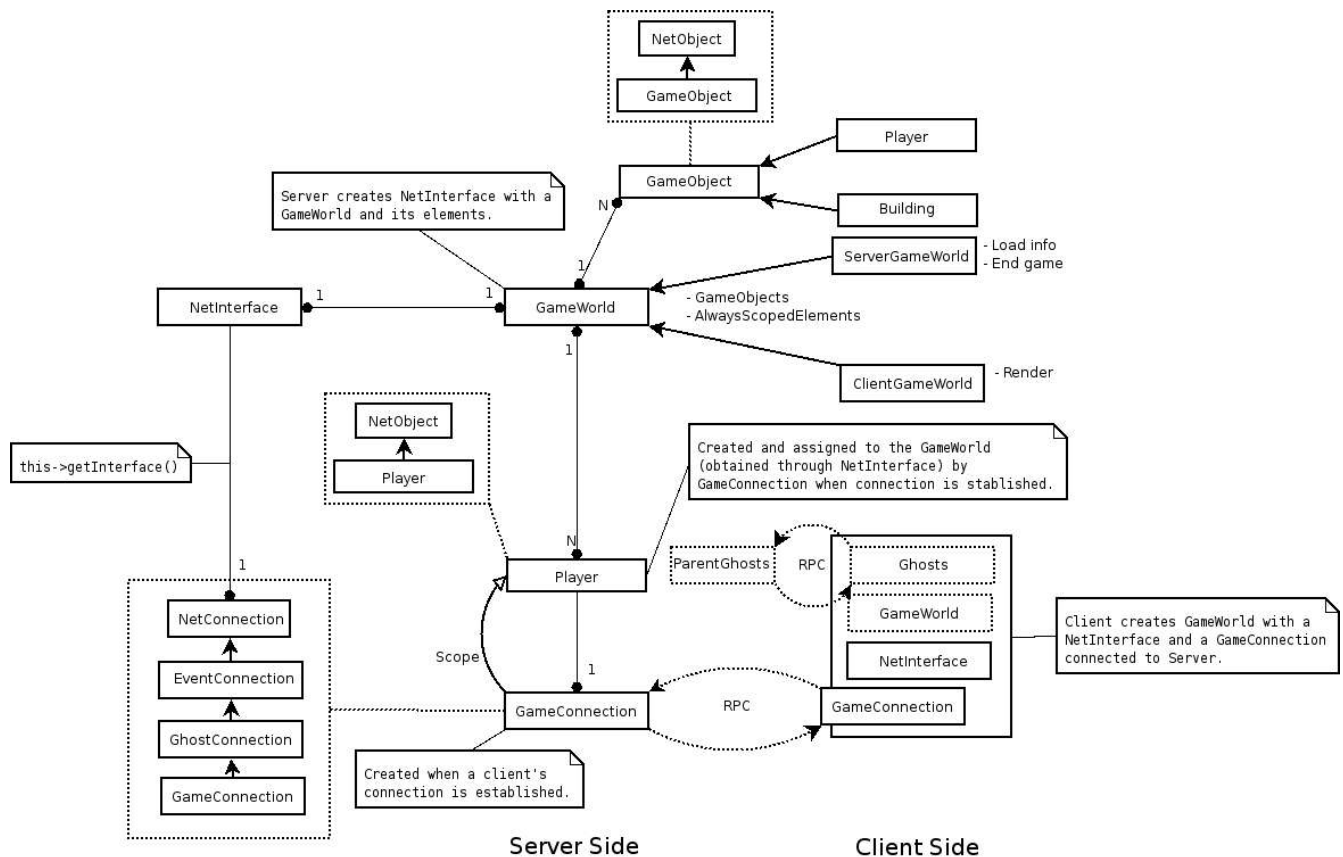


Demo, 3D world:

We can move our player, clicking in any place of the terrain.

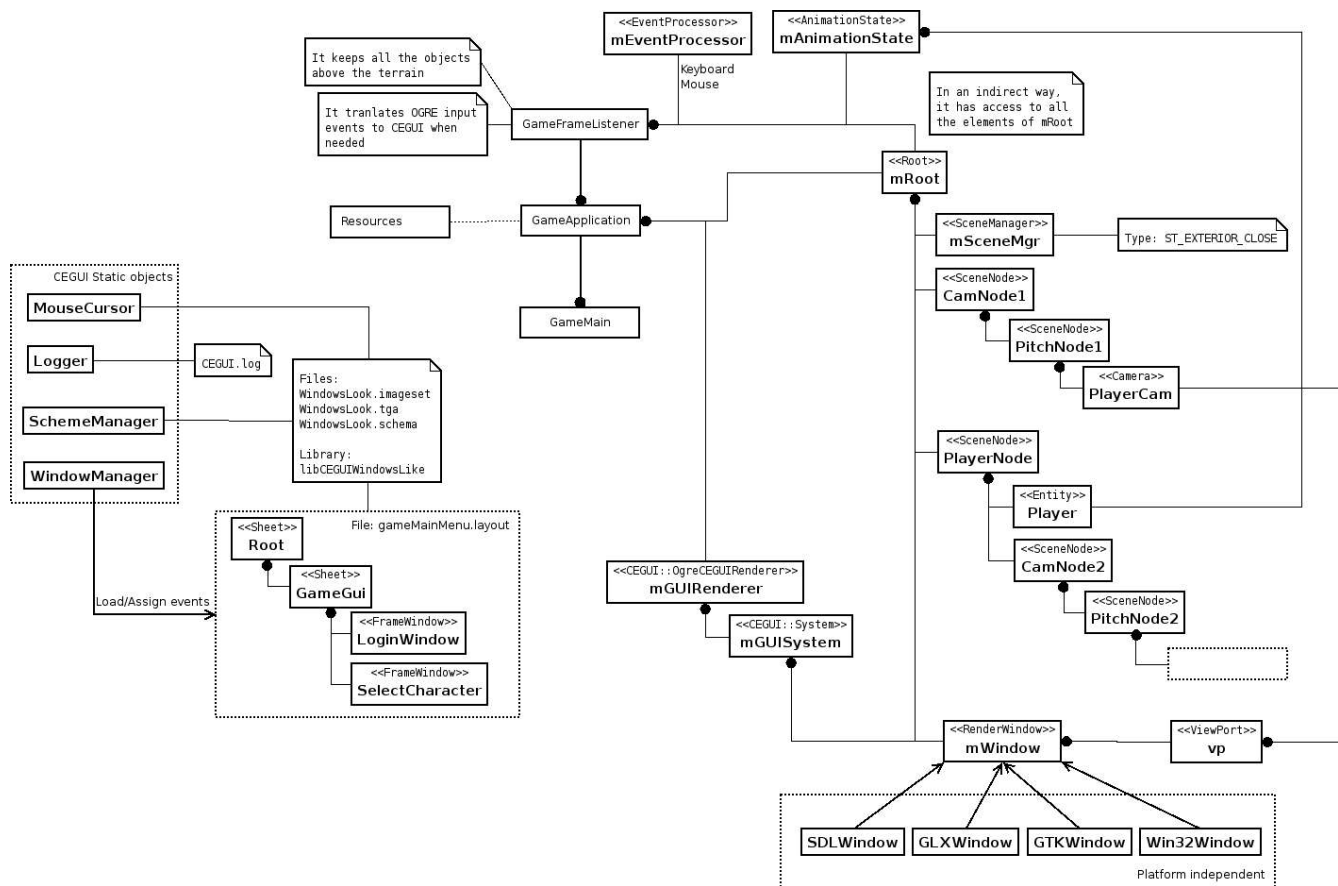


## Networking Layer





## 3D Interface Layer



## Suggestions for further work

In order to continue the work, next steps can be:

- Try to solve the problems with OGRE's Memory Manager and OpenTNL in MS Windows.
- Define and implement a good terrain division with tiles.
- Map objects to this tiles (for example, houses).
- Identify to which hexagon belongs a clicked pixel.
- Make our character find its way avoiding obstacles (for example, houses).
- Implement virtual character with reproductive capabilities using genetic algorithms.
- Experiment with the evolution of virtual character. How can a player interact with that characters? Can it give more fun to the game?

## Coordinates in Hexagon based tile maps

Thomas Jahn

Tile Maps are often used for the graphics in two-dimensional Games. Using Tiles means that all graphics are combinations of smaller graphics, similar to mosaics. The advantage is a smaller need for Memory. The Tilesets are seldom bigger than some

hundreds of Kilobytes and a Map needs far less memory than a pre rendered image; it needs only a pointer per field that identifies the used Tile.

The simplest tiling system has plain squares. It's used in games like Warcraft II or the first C&C parts.

For isometric views a la Diablo or Ultima7 you usually use diamond shaped Tiles. These Tiles allow you to achieve a pseudo 3D look.

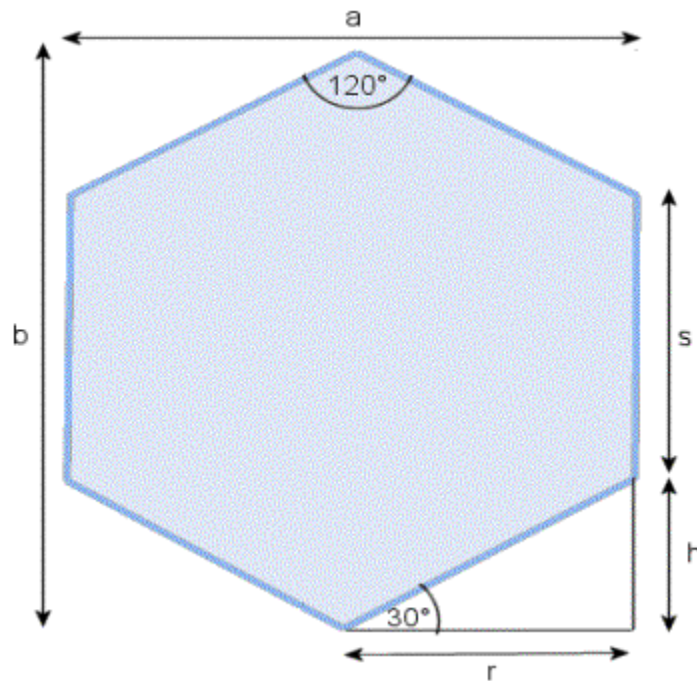
Both tiling systems have a major disadvantage. All movements are restricted to four directions (vertical and horizontal for squares and diagonal for diamonds) or you have to cope with different step distances. (The distance between two tiles that touch only at the edges is bigger, then the distance of tiles that touch at the sides.)

A very nice solution offers a hexagon based tile system. It allows movements in 6 directions with equal distances, so it's used especially by tactical games like Panzer General. A second advantage is that those hexagon tiled maps look just better then simple square maps.

On the other side it is harder to handle for artists as well as for the programmers.

## The mathematical structure of an hexagon

First of all let's have a look at the mathematical characteristics of a hexagon.



In a hexagon all six sides have the same length **s** that meet in an angle of  $120^\circ$ .

All other values depend on **s**, so if you set **s** you can easily calculate the missing information:

The height **h**:

$$h = \sin(30^\circ) * s$$

The distance **r**:

$$r = \cos(30^\circ) * s$$

The height of the surrounding rectangle **b**:

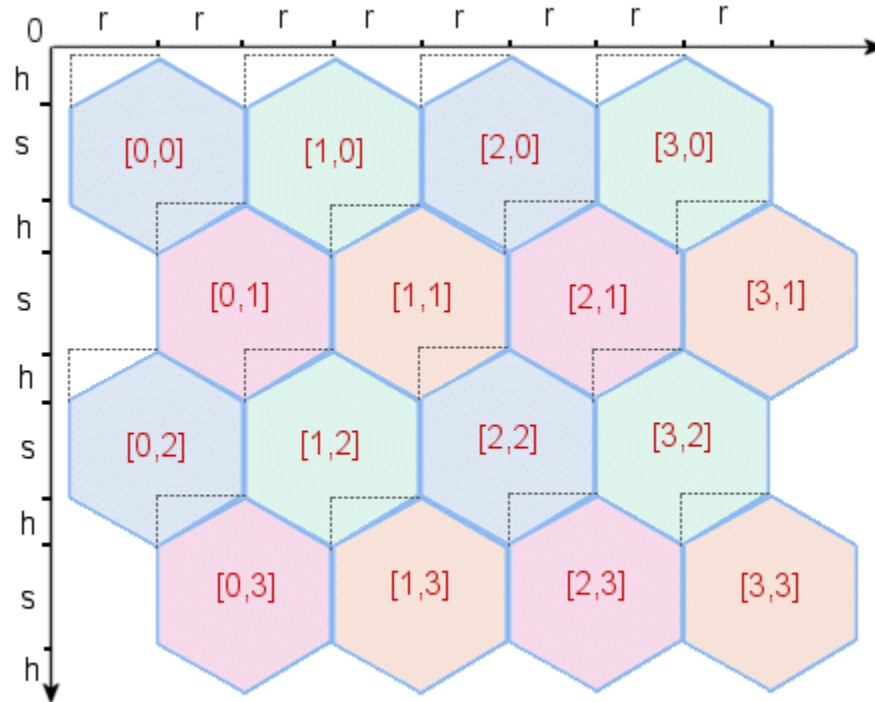
$$b = s + 2 * h$$

The width of the surrounding rectangle **a**:

$$a = 2 * r$$

## Converting array coordinates to pixel coordinates

As in all other tile maps the information of hexagon based tile maps is stored in a two dimensional array. To plot hexagonal tiles we need to know the pixel coordinate of the top left edge of the rectangle that surrounds the hexagon. As we only know the array coordinate (the position in the array) we have to find a way to convert these coordinates to pixel coordinates.



If you have a look at the above picture you'll probably realize that every second row of tiles is indented by **r**.

For even rows (blue-green) we have to use this calculation:

$$x_{\text{pixel}} = x_{\text{field}} * 2 * r$$

$$y_{\text{pixel}} = y_{\text{field}} * (h + s)$$

For odd rows (red-orange) we have to add **r** to the horizontal pixel coordinate:

$$x_{\text{pixel}} = x_{\text{field}} * 2 * r + r$$

$$y_{\text{pixel}} = y_{\text{field}} * (h + s)$$

Pseudocode:

```
PixelX := ArrayX * 2 * TileR + (FieldX AND 1) * TileR
PixelY := ArrayY * (HexagonH + HexagonS);
```

## Converting pixel coordinates to array coordinates

Now it should be quite easy to plot our map. But that's not enough! It becomes complicated again if we want to know on which tile (the array coordinate) our mouse cursor points. To solve this we have to find a way to convert pixel coordinates to array coordinates. A relatively simple solution is using a Mousemap (like CivII does) but there is a more elegant and faster way.

The first thing we do is to get the pixel coordinate. If we have scrollable map we have to add the position of the left top edge of our viewable sector to our cursor position.

Now we divide our map into rectangular sections. Have a look at the illustration below! It's quite easy to calculate in which section

our pixel coordinate lies in. A section has a height of  $h + s$  and a width of  $2 * r$  (or simply  $a$ ). Therefore we can make these simple equations:

$$x_{\text{section}} = x_{\text{pixel}} / (2 * r)$$

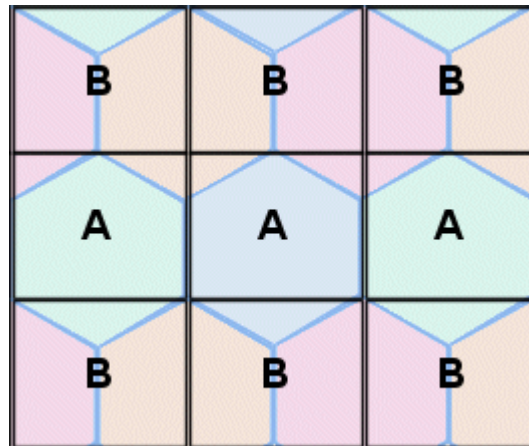
$$y_{\text{section}} = y_{\text{pixel}} / (h + s)$$

Keep in mind that you have to round off when doing these calculations.

It's not enough to know in which section we are. We also need to know on which pixel within the section we are. To get that we subtract the position of the left top edge of our section from the global pixel coordinate.

$$x_{\text{sectpxl}} = x_{\text{pixel}} - x_{\text{section}} * (2 * r)$$

$$y_{\text{sectpxl}} = y_{\text{pixel}} - y_{\text{section}} * (h + s)$$



There are two different types of section as you can see. It's very important to know of what type our section is. Looking at the above illustration you might realize that sections in even rows (those with an even  $y_{\text{section}}$  value) are type A while sections in odd rows are of type B.

Pseudocode:

```
SectX := PixelX div (2 * TileR);
SectY := PixelY div (HexagonH + HexagonS);

SectPxLX := PixelX mod (2 * TileR);
SectPxLY := PixelY mod (HexagonH + HexagonS);

If (SectY AND 1 = 0) then SectTyp := A else SectTyp := B;
```

Now we have all information we need to determine the array coordinate of the tile our pixel is in. Depending on what section we have our further calculations are different.

### A-Sections

A-Sections consist of three areas.

If the pixel position in question lies within the big bottom area the array coordinate of the tile is the same as the coordinate of our section.

If the position lies within the top left edge we have to subtract one from the horizontal (x) and the vertical (y) component of our section coordinate.

If the position lies within the top right edge we reduce only the vertical component.

Every edge has a gradient of either

$$m = \text{HexagonH} / \text{HexagonR} \text{ or } m' = - \text{HexagonH} / \text{HexagonR}.$$

Pseudocode

```
// middle
ArrayY := SectY;
ArrayX := SectX;

// left Edge
if SectPxLY < (HexagonH - SectPxLX * m) then begin
  ArrayY := SectY - 1;
  ArrayX := SectX - 1;
end;

// right Edge
if SectPxLY < (- HexagonH + SectPxLX * m) then begin
  ArrayY := SectY - 1;
  ArrayX := SectX;
end;
```

## B-Sections

B-Sections consist of three areas, too. But they are shaped differently!

If the pixel position in question lies within the right area the array coordinate of the tile is the same as the coordinate of our section.

If the position lies within the left area we have to subtract one from the horizontal (x) component of our section coordinate.

If the position lies within the top area we have to subtract one from the vertical (y) component.

Pseudocode:

```
// right side
if SectPxLX >= HexagonR then begin
  if sectPxLY < (2 * HexagonH - SectPxLX * m) then begin
    ArrayY := SectY - 1;
    ArrayX := SectX;
  end else begin
    ArrayY := SectY;
    ArrayX := SectX;
  end;
end;

// left side
if SectPxLX < HexagonR then begin
  if SectPxLY < (SectPxLX * m) then begin
    ArrayY := SectY - 1;
    ArrayX := SectX;
  end else begin
    ArrayY := SectY;
    ArrayX := SectX - 1;
  end;
end;
```

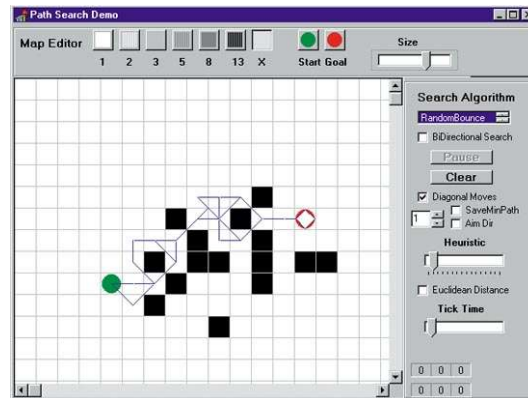
## Path finding

Smart Moves: Intelligent Path-Finding  
W. Bryan Stout

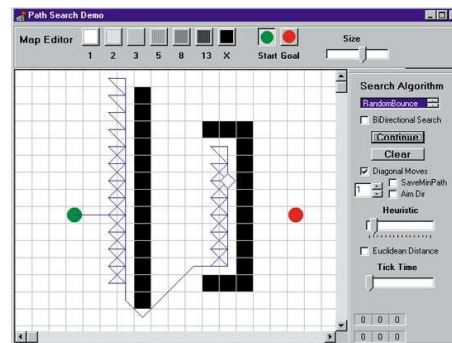
The main problem in order to find a good path between two points, is avoiding obstacle. There are several solution to this situation:

### Movement in a random direction.

If the obstacles are all small and convex, the entity (shown as a green dot) can probably get around them by moving a little bit away and trying again, until it reaches the goal (shown as a red dot).

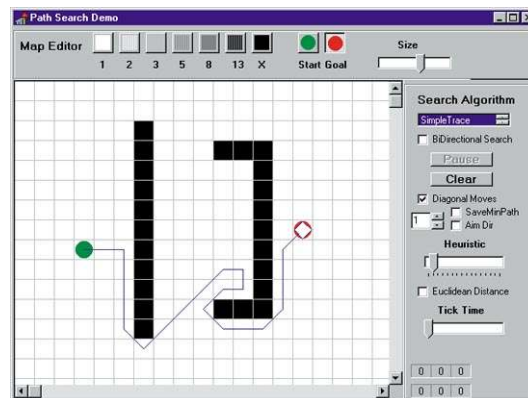


A problem arises with this method if the obstacles are large or if they are concave, the entity can get completely stuck, or at least waste a lot of time before it stumbles onto a way around. One way to avoid this: if a problem is too hard to deal with, alter the game so it never comes up. That is, make sure there are never any concave obstacles.

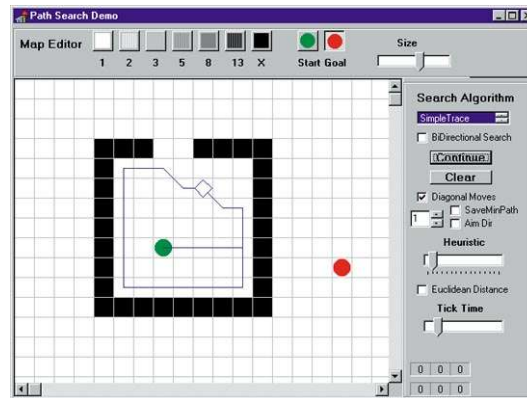


### Tracing around the obstacle.

Fortunately, there are other ways to get around. If the obstacle is large, one can do the equivalent of placing a hand against the wall and following the outline of the obstacle until it is skirted.

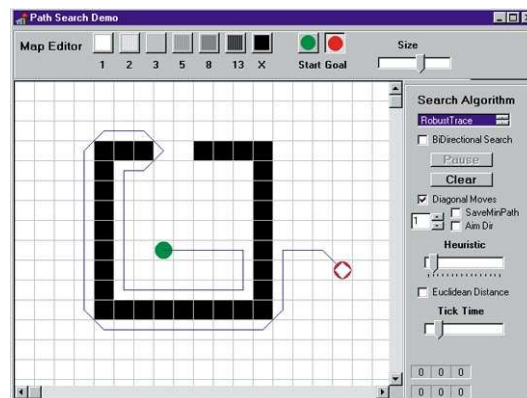


The problem with this technique comes in deciding when to stop tracing. A typical heuristic may be: "Stop tracing when you are heading in the direction you wanted to go when you started tracing." This would work in many situations, but one may end up constantly circling around without finding the way out.

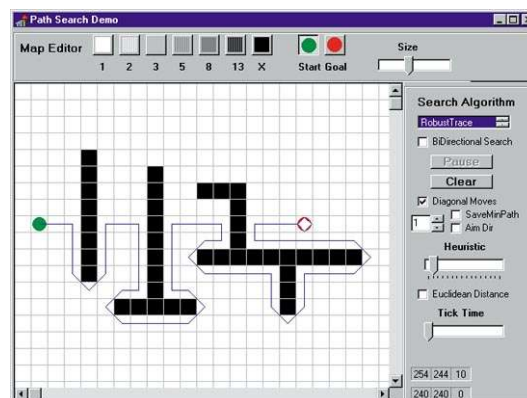


### Robust tracing.

A more robust heuristic comes from work on mobile robots: "When blocked, calculate the equation of the line from your current position to the goal. Trace until that line is again crossed. Abort if you end up at the starting position again." This method is guaranteed to find a way around the obstacle if there is one (If the original point of blockage is between you and the goal when you cross the line, be sure not to stop tracing, or more circling will result).



The downside of this approach: it will often take more time tracing the obstacle than is needed, making it look pretty simple-minded-though not as simple as endless circling. A happy compromise would be to combine both approaches: always use the simpler heuristic for stopping the tracing first, but if circling is detected, switch to the robust heuristic.



Although the obstacle-skirting techniques discussed above can often do a passable or even adequate job, there are situations where the only intelligent approach is to plan the entire route before the first step is taken. In addition, these methods do little to handle the problem of weighted regions, where the difficulty is not so much avoiding obstacles as finding the cheapest path among several choices where the terrain can vary in its cost.

Fortunately, the fields of Graph Theory and conventional AI have several algorithms that can be used to handle both difficult obstacles and weighted regions. In the literature, many of these algorithms are presented in terms of changing between states, or

traversing the nodes of a graph. They are often used in solving a variety of problems, including puzzles like the 15-puzzle or Rubik's cube, where a state is an arrangement of the tiles or cubes, and neighbouring states (or adjacent nodes) are visited by sliding one tile or rotating one cube face. Applying these algorithms to path-finding in geometric space requires a simple adaptation: a state or a graph node stands for the entity being in a particular tile, and moving to adjacent tiles corresponds to moving to the neighbouring states, or adjacent nodes.

Working from the simplest algorithms to the more robust.

## Breadth-first search

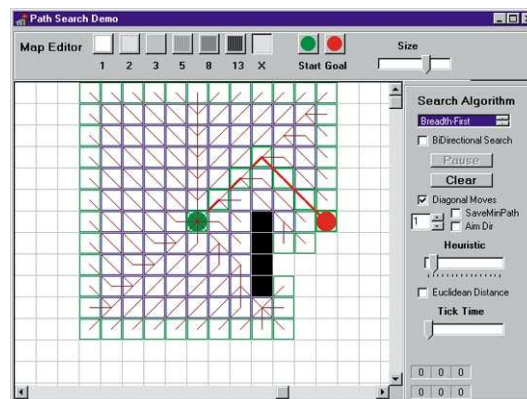
Beginning at the start node, this algorithm first examines all immediate neighbouring nodes, then all nodes two steps away, then three, and so on, until a goal node is found. Typically, each node's unexamined neighbouring nodes are pushed onto an Open list, which is usually a FIFO (first-in-first-out) queue. The algorithm would go something like:

```

queue          Open
BreadthFirstSearch
  node  n, n', s
  s.parent = null
  // s is a node for the start
  push s on Open
  while Open is not empty
    pop node n from Open
    if n is a goal node
      construct path
      return success
    for each successor n' of n
      if n' has been visited already,
        continue
      n'.parent = n
      push n' on Open
  return failure // if no path found

```

This picture shows how the search proceeds:

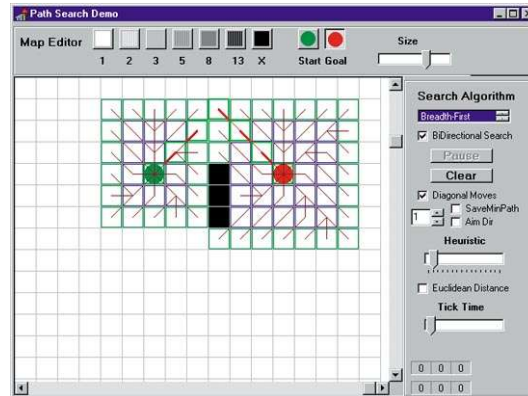


We can see that it does find its way around obstacles, and in fact it is guaranteed to find a shortest path—that is, one of several paths that tie for the shortest in length—if all steps have the same cost. There are a couple of obvious problems. One is that it fans out in all directions equally, instead of directing its search towards the goal; the other is that all steps are not equal—at least the diagonal steps should be longer than the orthogonal ones.

## Bidirectional breadth-first search

This enhances the simple breadth-first search by starting two simultaneous breadth-first searches from the start and the goal nodes and stopping when a node from one end's search finds a neighbouring node marked from the other end's search.





This can save substantial work from simple breadth-first search (typically by a factor of 2), but it is still quite inefficient. Tricks like this are good to remember, though, since they may come in handy elsewhere.

## Dijkstra's algorithm.

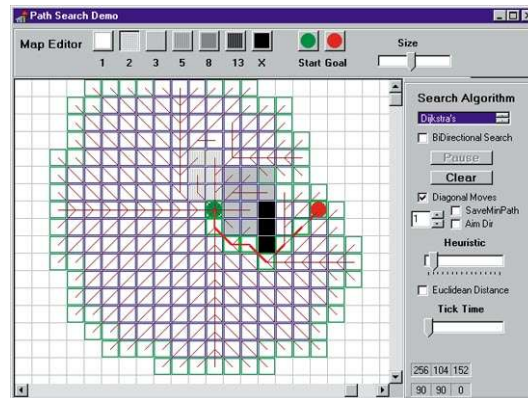
E. Dijkstra developed a classic algorithm for traversing graphs with edges of differing weights. At each step, it looks at the unprocessed node closest to the start node, looks at that node's neighbours, and sets or updates their respective distances from the start. This has two advantages to the breadth-first search: it takes a path's length or cost into account and updates the goodness of nodes if better paths to them are found. To implement this, the Open list is changed from a FIFO queue to a priority queue, where the node popped is the one with the best score-here, the one with the lowest cost path from the start.

```

priority queue Open

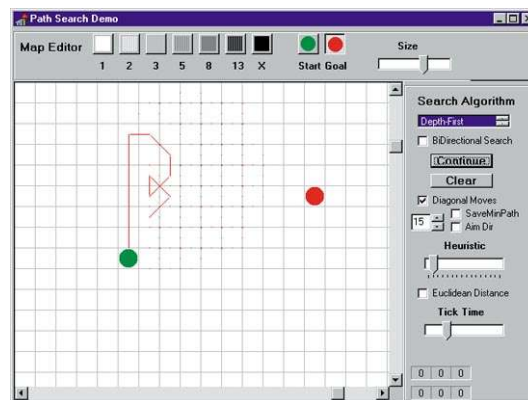
DijkstraSearch
  node  n, n', s
  s.cost = 0
  s.parent = null  // s is a node for the start
  push s on Open
  while Open is not empty
    pop node n from Open  // n has lowest cost in Open
    if n is a goal node
      construct path
      return success
    for each successor n' of n
      newcost = n.cost + cost(n,n')
      if n' is in Open
        and n'.cost <= newcost
          continue
      n'.parent = n
      n'.cost = newcost
      if n' is not yet in Open
        push n' on Open
  return failure  // if no path found
  
```

In the next picture we see that Dijkstra's algorithm adapts well to terrain cost. However, it still has the weakness of breadth-width search in ignoring the direction to the goal.



## Depth-first search

This search is the complement to breadth-first search; instead of visiting all a node's siblings before any children, it visits all of a node's descendants before any of its siblings. To make sure the search terminates, we must add a cutoff at some depth. We can use the same code for this search as for breadth-first search, if we add a depth parameter to keep track of each node's depth and change Open from a FIFO queue to a LIFO (last-in-first-out) stack. In fact, we can eliminate the Open list entirely and instead make the search a recursive routine, which would save the memory used for Open. We need to make sure each tile is marked as "visited" on the way out, and is unmarked on the way back, to avoid generating paths that visit the same tile twice. The next picture shows that we need to do more than that: the algorithm still can tangle around itself and waste time in a maddening way.



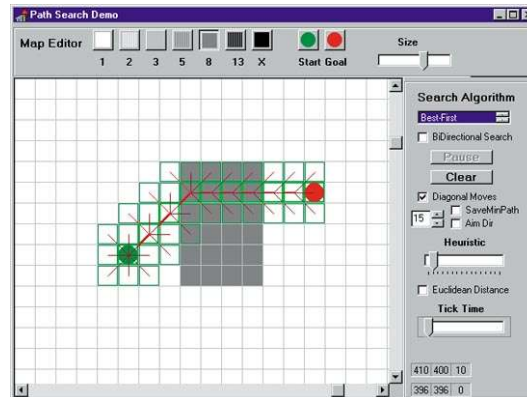
For geometric path-finding, we can add two enhancements. One would be to label each tile with the length of the cheapest path found to it yet; the algorithm would then never visit it again unless it had a cheaper path, or one just as cheap but searching to a greater depth. The second would be to have the search always look first at the children in the direction of the goal. With these two enhancements checked, one sees that the depth-first search finds a path quickly. Even weighted paths can be handled by making the depth cut-off equal the total accumulated cost rather than the total distance.

## Iterative-deepening depth-first search

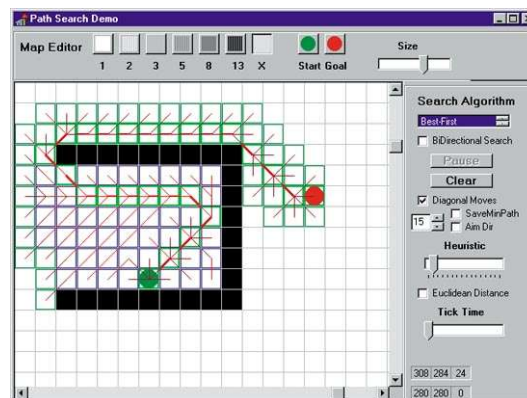
Actually, there is still one fly in the depth-first ointment-picking the right depth cutoff. If it is too low, it will not reach the goal; if too high, it will potentially waste time exploring blind avenues too far, or find a weighted path which is too costly. These problems are solved by doing iterative deepening, a technique that carries out a depth-first search with increasing depth: first one, then two, and so on until the goal is found. In the path-finding domain, we can enhance this by starting with a depth equal to the straight-line distance from the start to the goal. This search is asymptotically optimal among brute force searches in both space and time.

## Best-first search

This is the first heuristic search considered, meaning that it takes into account domain knowledge to guide its efforts. It is similar to Dijkstra's algorithm, except that instead of the nodes in Open being scored by their distance from the start, they are scored by an estimate of the distance remaining to the goal. This cost also does not require possible updating as Dijkstra's does.



It is easily the fastest of the forward-planning searches we have examined so far, heading in the most direct manner to the goal. We also see its weaknesses. In 8A, we see that it does not take into account the accumulated cost of the terrain, plowing straight through a costly area rather than going around it. And in the next picture, we see that the path it finds around the obstacle is not direct, but weaves around it in a manner reminiscent of the hand-tracing techniques seen above.



## A\* Search

The best-established algorithm for the general searching of optimal paths is A\*. This heuristic search ranks each node by an estimate of the best route that goes through that node. The typical formula is expressed as:

$$f(n) = g(n) + h(n)$$

where:

- $f(n)$  is the score assigned to node  $n$
- $g(n)$  is the actual cheapest cost of arriving at  $n$  from the start
- $h(n)$  is the heuristic estimate of the cost to the goal from  $n$

It combines the tracking of the previous path length of Dijkstra's algorithm, with the heuristic estimate of the remaining path from best-first search.

priorityqueue    Open

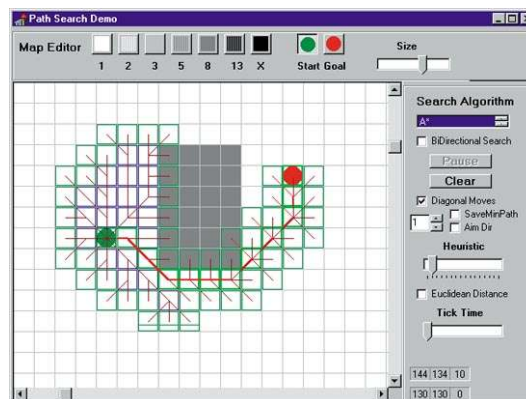
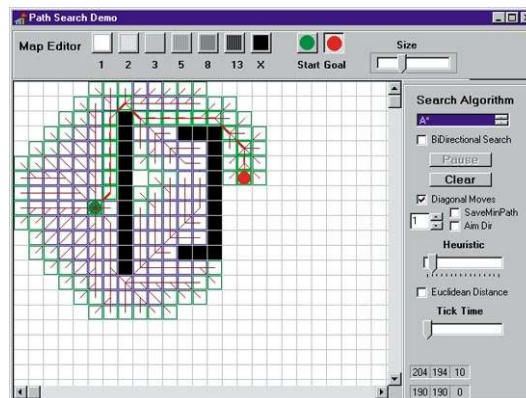
```

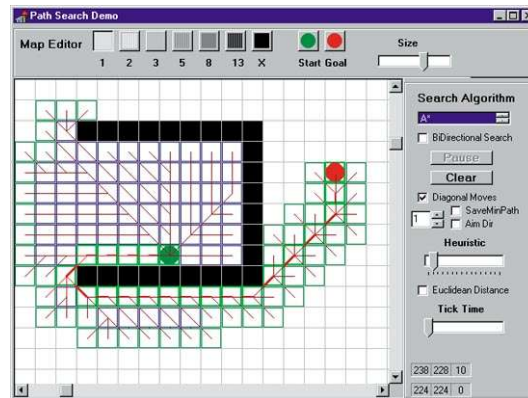
list          Closed
AStarSearch
    s.g = 0          // s is the start node
    s.h = GoalDistEstimate( s )
    s.f = s.g + s.h
    s.parent = null
    push s on Open
    while Open is not empty
        pop node n from Open // n has the lowest f
        if n is a goal node
            construct path
            return success
        for each successor n' of n
            newg = n.g + cost(n,n')
            if n' is in Open or Closed,
                and n'.g <= newg
                skip
            n'.parent = n
            n'.g = newg
            n'.h = GoalDistEstimate( n' )
            n'.f = n'.g + n'.h
            if n' is in Closed
                remove it from Closed
            if n' is not yet in Open
                push n' on Open
        push n onto Closed
    return failure // if no path found

```

Since some nodes may be processed more than once-from finding better paths to them later-we use a new list called Closed to keep track of them.

A\* has a couple interesting properties. It is guaranteed to find the shortest path, as long as the heuristic estimate,  $h(n)$ , is admissible-that is, it is never greater than the true remaining distance to the goal. It makes the most efficient use of the heuristic function: no search that uses the same heuristic function  $h(n)$  and finds optimal paths will expand fewer nodes than A\*, not counting tie-breaking among nodes of equal cost. In the next pictures, we see how A\* deals with situations that gave problems to other search algorithms.





The quality of A\*'s search depends on the quality of the heuristic estimate  $h(n)$ . If  $h$  is very close to the true cost of the remaining path, its efficiency will be high; on the other hand, if it is too low, its efficiency gets very bad. In fact, breadth-first search is an A\* search, with  $h$  being trivially zero for all nodes-this certainly underestimates the remaining path cost, and while it will find the optimum path, it will do so slowly.

## References

BOOK: Design Patterns (ISBN 0201633612)

Authors: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Gang of Four, GoF)

PAPER: Game Design Patterns

Authors: Staffan Björk, Sus Lundgren, Jussi Holopainen

PAPER: Design Patterns for Games

Authors: Dung ("Zung") Nguyen and Stephen B. Wong

PAPER: A model to support the design of Multiplayer Games

Authors: José Pablo Zagai, Miguel Nussbaum, Ricardo Rosas

PAPER: Aspects of Networking in Multiplayer Computer Games

Authors: Jouni Smed, Timo Kaukoranta, Harri Hakonen

PAPER: Human-like Behaviour in Real-Time Strategy Games: An Experiment With Genetic Algorithms

Authors: Fredrik Olofsson, Johan W. Andersson

PAPER: Using Genetic Algorithms for Game AI

Author: Greg James

PAPER: Social Networking in Massively Multiplayer Online Games

Authors: Mikael Jakobsson, T.L. Taylor

Article: Men are from Quake, Women are from Ultima I

Emily Laber (NY Time article)

Documentation OpenTNL

<http://www.opentnl.org/docs.php>

Documentation OGRE 3D

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

Article: Coordinates in Hexagon based Tile Maps

Thomas Jahn

## Other Resources

CiteSeer, <http://citeseer.ist.psu.edu/>  
DevMaster, <http://www.devmaster.net/>  
GameDev, <http://www.gamedev.net/>  
Wikipedia, <http://en.wikipedia.org>