AQUARIUS

Technical Design Document
Version 1.0

Team Thalassic

Contents

1 Document Conventions

1.1 Highlighted Text and other Caveats

Highlighted Text refers to a Higher Goal. When un-Highlighted text is followed by contradictory Highlighted Text, the functionality described by the Highlighted Text will eventually supersede the functionality described by the un-Highlighted text. While the Project Planning Timeline plan for implementing everything described in this document, all Highlighted Text can be cut from the design at any time for any reason.

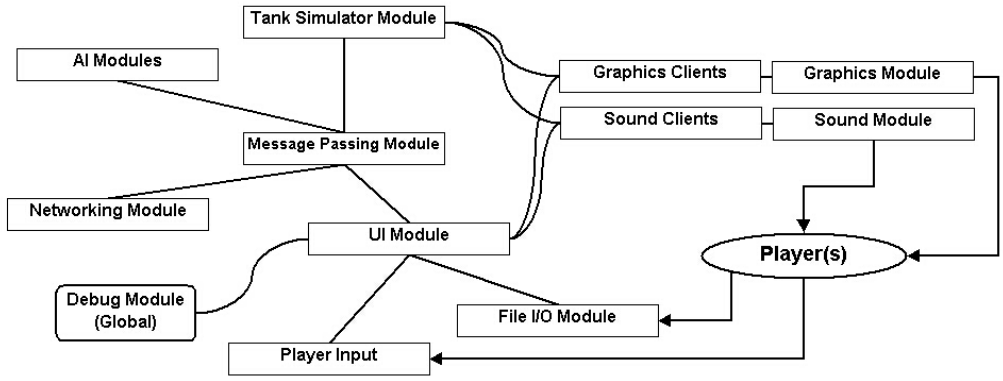
2 Introduction

Project Aquarius is an evolutionary simulation set in a 3D aquatic environment. The primary focus of the project is to explore a variety of artificial intelligence techniques by allowing them to run in a genetic algorithm. The vehicle for selection of this algorithm is survival in the aquatic environment.

2.1 Module overview

The code is divided into encapsulated modules that communicate via the message handler. The modules are briefly described below. Detailed descriptions are available in the modules corresponding TDD section.

High Level Object Heirarchy



Message Handler All inter-module communication goes through the Message Handler. This message handler will make Networking invisible to the system by routing messages to objects not on the current computer to other connected computers. The Message Handler is the central nervous system of the program.

Networking The networking module will be invisible to the system since all messages will be sent via the message handler. It will employ UDP packets to transmit messages to connected computers.

AI The AI system will handle decision making for fish within the system as well as evolving fish within the system.

Tank The tank is the ecosystem manager. It manages terrain, physics, fish bodies, plankton, current and other objects that may appear in the game environment.

Physics The physics module will handle object physical interactions in the game environment including collisions, buoyancy and fluid movement.

Graphics The graphics module will handle all graphics using OpenGL. It will handle preparing the window for use with OpenGL.

Sound The sound module will handle all sound effects and streaming sounds within the game.

Menu System The menu system will consist of GUI primitives that will allow the construction of all interface screens within the game.

Support Services In addition to the main systems, there will be several support systems available to assist the main systems.

Threading The threading module will provide support for multi-threading.

Debugger The debugger will provide support for displaying messages that are used by the programmers to track errors or to ensure that the code is working properly.

File I/O The file I/O manager will provide support for saving and loading data to files.

Player Input The player input module will poll devices for input from the user. It will make this information available to the system.

Math The math module will include classes for points, vectors as well as common linear algebra operations.

Timing Object Will provide information regarding time, frame rate and time steps.

2.2 Global Data

Game State The state of the game will be available as global data. Possible states are online, offline and pause.

Online	Standard game mode
Offline	UI screen displayed in place of 3D tank graphics. Simulation progresses at an accelerated rate.
Pause	Pauses the game.

Windows HINSTANCE The handle to the windows instance will be made globally available for external libraries that may need this information.

Timing Object Since several objects will need time updates every frame, this is made globally available for efficiency.

Debugger The debugger object will be made globally available to reduce debugger dependencies.

2.3 Program Flow

2.3.1 Start up

The order of object initialization is as follows:

1. Graphics
2. Sound
3. Message Handler
4. Support Systems
5. Networking
6. Tank
7. AI
8. UI

2.3.2 Main loop

When the game starts it will initially be in the pause state. During this time the user interface will go through its intro screens. After the intro screens are done the game will go into online mode.

Timer.logic	Gets current time and performs frame rate calculations
Networking.logic	Network messages are sent and received
UI.logic	Input devices are polled
	User input is processed
AI logic	AI processing
AI.breed	Genetic Algorithm is processed
Tank.logic	Environment, fish and physics processing
Sound.play	Sound module processing
Graphics.draw	Graphics module processing

2.3.3 Shut down

The system is shut down in the reverse order of start up.

3 Message Handler

The Message Handler has several purposes:

1. It will decouple the data that defines the current state of the Environment (the Tank Simulator) from the operations that nondeterministically modify that data (Player Input and AI [and Networking, which acts as a surrogate for remote Player Input and AI]).
2. It ensures that the Networking, Player Input, AI and the Tank Simulator are able to use a simple interface to exchange data with the rest of the system.
3. It hands out, and keeps track of, the numbers that uniquely identify every Fish and Object in the Aquarium.

Dependencies
Tank Simulator
Networking
UI
AI

Patterns To Be Applied
Mediator
Singleton

Implementation The MessageHandler's unique 32-bit Key generation will be encapsulated as a private function (GenerateKey()); the associated data will be private as well. The most significant bit of the 32-bit key will be set if the Key is Official (that is, generated by a Server); the most significant bit will not be set if the Key is Unofficial (that is, generated by a Client). A complementary ReturnKey() function is provided to "free" a Key so that it can be used to identify a different entity. unsigned int is typedef'd to be Key.

The MessageHandler will have a private function that will allow it to ValidateKey() (which throws an exception if passed an illegal Key) to localize bugs resulting from illegal Keys. This function could be removes late in development for a small speed increase if necessary.

The MessageHandler will keep pointers to the Networking, AI, UI and Tank Simulator modules. It also provides public functions that allow each of these modules, in their constructors, to "register" with the MessageHandler by passing their "this" pointers – in exchange, the MessageHandler will provide a MessageHandler reference to the "registering" module. It will most certainly crash if one were to attempt to use its functionality before these pointers were initialized.

The MessageHandler is a monolithic structure that encapsulates a series of functions that correspond to every message that the AI, Networking, UI and

Tank Simulator modules can send to one another. It is entirely reactive – that is, it never initiates any action by itself, but instead Mediates interactions between the AI, Networking, UI and Tank Simulator modules.

Although the public functions of the MessageHandler will be defined by the modules it Mediates (and will most definitely evolve and change as development progresses), there are several basic kinds of functions that the MessageHandler will provide. Examples follow:

1. Functions that do not need to be sent over the Network.

```
unsigned int GetFishHealth(Key FishKey, int &FishHealth)
{
    ValidateKey(FishKey);
    return tankSimulator->TankSimulator::GetFishHealth( FishKey,
                                                        FishHealth);
}
```

2. Functions that need to be sent over the Network.

```
void MoveFish(Key FishKey, FishMovementData &FishMovement)
{
    ValidateKey(FishKey);
    tankSimulator->TankSimulator::MoveFish(FishMovement);

    GameNetworking::NetMessage netMsg;
    netMsg.SetID(GameNetworking::FISH_MOVEMENT);
    netMsg.Write4Bytes(FishMovement);
    networking->GameNetworking::SendNetMessage(netMsg);

    return;
}
```

3. Functions that require the MessageHandler to generate a new Key.

```
void CreateNewFish(FishData &fishData)
{
    fishData.Key = GenerateNewKey();
    tankSimulator->TankSimulator::CreateNewFish(fishData);

    GameNetworking::NetMessage netMsg;
    netMsg.SetID(GameNetworking::CREATE_NEW_FISH);
    netMsg.Write4Bytes(fishData);
    networking->GameNetworking::SendNetMessage(netMsg);

    return;
}
```

4. Functions that return a Key to the MessageHandler.

```
void DestroyFish(FishData &fishData)
{
    ValidateKey(FishKey);
    ReturnKey(fishData.Key);
    tankSimulator->TankSimulator::DestroyFish(fishData);

    GameNetworking::NetMessage netMsg;
    netMsg.SetID(GameNetworking::DESTROY_FISH);
    netMsg.Write4Bytes(fishData);
    networking->GameNetworking::SendNetMessage(netMsg);

    return;
}
```

```
class MessageHandler
{
private:
    //Key generation data
    //ValidateKey(), GenerateNewKey(), ReturnKey()
    //pointers to Networking, AI, UI and TankSimulator modules
public:
    RegisterNetworking(), RegisterAI(), etc. functions
    //Lots of functions that define intercommunications between
    //the Networking, AI, UI and TankSimulator modules.  Some
    //examples in addition to the above
    CollectServerUpdate()
    InstituteServerUpdate()
    JoinServer()
    FishBite()
    //...and so forth
};
```

Concerns The MessageHandler module is a potential bottleneck. Each function in the MessageHandler is essentially overhead, and therefore must execute with the utmost speed.

Failure to return Keys will eventually cause the game to choke, as the MessageHandler endlessly searches for an unused Key.

4 Networking

The Networking Module is intended to be used to send and receive well-defined unreliable data packets between computers remotely participating in a game.

4.1 UDPNetworking

The UDPNetworking class will be a collection of public functions that wrap the WinSock 2.2 API. It will provide the low level Networking functionality that the GameNetworking class will require.

Dependencies
None

Patterns To Be Applied
Singleton

Implementation The IP Address and Host Name of the local machine will be encapsulated as private data. This frees clients of UDPNetworking from having to know about their own local address information.

UDPNetworking's constructor will start up WinSock 2.2 system. It will initialize the private local address information. Throws an exception on failure.

CreateSock() takes a pointer to a SOCKET and a port number (which can be ANY_PORT if the user does not care which port the SOCKET resides on). CreateSock() either sets the SOCKET pointer to point to a valid SOCKET or throws an exception.

DestroySock() takes a SOCKET. It either frees the resources associated with this SOCKET or throws an exception.

SendPacket() takes the SOCKET to send from, the port and IP address to send to, a **void** $*$ to the data to be sent, and the size, in bytes of, the data to be sent. It throws an exception if the size of the data to be sent is greater than MAX_PACKET_SIZE or on failure – otherwise it sends the specified data from the specified SOCKET to the specified port at the specified IP Address.

ReceivePacket() takes the socket to receive on, a pointer to the buffer that is to hold the received data, the size of the buffer that is to hold the received data, a pointer to an IP Address and a pointer to a Port. ReceivePacket() will not return until data becomes available to be read from the specified socket. If ReceivePacket() returns, the received data is placed in the specified buffer, the IP Address pointer will point to the IP Address of the sender, and the Port pointer will point to the port number the sender sent from. If an error is encountered, ReceivePacket() throws an exception.

MyIPAddress() takes no parameters and returns the local machine's IP Address.

UDPNetworking's destructor will shut down the WinSock 2.2 system. Any SOCKETs that remain open will be closed, and any data queued up to be received on them will be discarded. Throws an exception on failure.

```
class UDPNetworking
{
private:
    //IP Address and Host Name of the local machine
public:
    UDPNetworking()
    ~UDPNetworking()
    CreateSock()
    DestroySock()
    SendPacket()
    ReceivePacket()
    MyIPAddress()
};
```

ANY_PORT will be a constant value defined in the UDPNetworking namespace. It can be passed as an argument to the Port parameter of CreateSock().

MAX_PACKET_SIZE will be a constant value defined in the UDPNetworking namespace – it is equal to the size, in bytes, of the largest possible "payload" of an unreliable data packet.

Concerns UDPNetworking is not a flawless abstraction of the WinSock 2.2 API. For example, it exposes the SOCKET datastructure to its clients, as well as propagating WinSock's notions of 2 byte unsigned integral Port numbers and 4 byte unsigned integral IP Addresses. This has the potential to cause confusion, especially in programmers not familiar with WinSock.

4.2 GameNetworking namespace

The GameNetworking namespace encompasses the classes that implement the networking logic the game requires. Many of these classes require that the UDPNetworking class has already been instantiated.

4.2.1 GameNetworking class

GameNetworking class is the base class for the Server and Client classes. It encapsulates functionality common to both subclasses.

Dependencies
UDPNetworking
MessageHandler

Patterns To Be Applied
Singleton

Implementation GameNetworking privately encapsulates a mutexed list of Events (Events contain NetMessages that are waiting to be sent to the MessageHandler).

GameNetworking privately encapsulates a Thread object that asynchronously retrieves Packets from the SOCKETs in ReceiveSocksToWaitOn. If the retrieved Packet has more than one TotalPage, the function scans through the list of Events, searching for an Event that contains a NetMessage that has the same DocumentNum as the current Packet – if one is found, the current Packet is NetMessage.ConcatenatePacket()'ed to that NetMessage. Otherwise the Packet either has only one TotalPage or no Event in the Event list contains a NetMessage that has the same DocumentNum as the Packet, and the Thread NetMessage.ConcatenatePacket()'s the Packet, places that NetMessage into an Event (timestamping the Event and recording the IP Address of the sender of the NetMessage), and places the Event in the mutexed list of Events.

GameNetworking privately encapsulates the method GenerateDocumentNum(), as well as the unsigned int curDocumentNum. curDocumentNum is initialized to 0 in GameNetworking's constructor, and each time GenerateDocumentNum() is called, the function simply increments curDocumentNum by 1 and returns it by value. The function is intended to be used to stamp an outgoing NetMessages' Packet(s) with the same DocumentNum, so that the Packets can be reassembled into a single NetMessage on the receiving end. Since curDocumentNum will only wrap back around to 0 after 2^{32} NetMessages have been sent, there is practically no danger of stamping two different NetMessage's Packets with the same DocumentNum.

GameNetworking contains a protected SOCKET for receiving packets (receiveSock), and the port at which receiveSock listens (receivePort).

GameNetworking contains the protected member object ReceiveSocksToWaitOn, which encapsulates one SOCKET for each remote player the GameNetworking is staying in contact with.

GameNetworking encapsulates, as protected members, a SOCKET reserved for sending packets (sendSock), a list of RemoteMachines that it is currently concerned with staying in contact with, a reference to the UDPNetworking object, and a reference to the MessageHandler object.

GameNetworking encapsulates the protected member function ProcessEvents(), which does a linear scan through the mutexed list of Events. An Event whose timestamp shows that it is "expired" (that is, a large NetMessage that was split into two or more Packets when it was sent has been only partially received – and at least one of the NetMessage's Packets has not yet been received in PACKET_TIMEOUT milliseconds) is removed from the list, and discarded (NetMessage.DeallocatePacket()'ed). An Event that contains a NetMessage which contains all of its Packets is taken off of the mutexed list. Next, the function makes note of the fact that a RemoteMachine just communicated to it (by finding the RemoteMachine whose IP Address matches the IP Address of the sender of the Event and setting RemoteMachine.timeSinceLastUpdate = 0), and finally NetMessage.GetID()'s in order to determine which MessageHandler function to pass the NetMessage to.

GameNetworking encapsulates the protected member function DetectRemoteMachineDelinquency(), which iterates through the list of RemoteMachines, and, whenever it finds a RemoteMachine that has not communicated in TIMEOUT milliseconds, it informs the MessageHandler.

GameNetworking's constructor and destructor are protected, to ensure that only derived classes can instantiate it. GameNetworking's constructor ReceiveSocksToWaitOn::AddSock()'s the ReceiveSock, and sets curDocumentNum to 0.

GameNetworking provides a public SendNetMessage() function that takes a NetMessage (by value). SendNetMessage() sets the first four bytes of each of the NetMessage's Packets to the same DocumentNum (so that the receiver can recognize that all of these Packets should be reassembled into a single NetMessage) by calling GenerateDocumentNum(), and then sends every Packet in the NetMessage to every RemoteMachine on the list of RemoteMachines by repeatedly calling UDPNetworking::SendPacket().

GameNetworking encapsulates the public method NotifySelf(), which takes a NetMessage (by value) as a parameter. It forwards the argument to UDPNetworking::SendPacket(), and fills out the IP Address argument with the local IP Address obtained from UDPNetworking::MyIPAddress() and the port argument with receivePort.

GameNetworking encapsulates the public member function BootRemoteMachine(), which takes the IP Address of the remote machine to disconnect. If the specified RemoteMachine is on the list of RemoteMachines, the function GameNetworking::SendNetMessage()'s a DISCONNECT message to that remote machine, ReceiveSocksToWaitOn::RemoveSocket()'s the SOCKET that was earmarked to receive messages from that remote machine and removes the corresponding RemoteMachine from the list of RemoteMachines, and returns true. If the specified Machine to boot is not on the list of RemoteMachines, the function returns false.

GameNetworking provides the public ConfigurePacketLoss() method, which takes the same arguments as ReceiveSocksToWaitOn()::ConfigurePacketLoss() – it simply forwards the arguments to ReceiveSocksToWaitOn()::ConfigurePacketLoss().

```
class GameNetworking
{
private:
    receiveSocket
    receivePort
    mutexed list of (incoming) Events
    Thread object
    GenerateDocumentNum()
    curDocumentNum
protected:
    ReceiveSocksToWaitOn object
    sendSocket
```

```
    MessageHandler reference
    UDPNetworking reference
    list of RemoteMachines
    ProcessEvents()
    DetectRemoteMachineDelinquency()
    GameNetworking()
    ~GameNetworking()
public:
    SendNetMessage()
    NotifySelf()
};
```

4.2.2 Thread Object

Thread is a function that runs in a separate thread of execution. It waits on the RemoteMachines specified by a GameNetworking object by using the ReceiveSocksToWaitOn object. When incoming data appears on the specified SOCKETs, the function receives the data, composes an Event from it, and pushes the Event onto the mutexed queue of Events in the GameNetworking object.

The Thread object's control flow is as follows:

- It is passed a reference to the ReceiveSocketsToWait object, a reference to the UDPNetworking object, and a reference to the list of Events upon launch of the Thread.
- Declares a local buffer of memory that is the same size as MAX_PACKET_SIZE.
- Loops infinitely on the following:
- Checks the boost thread property to see if RemoveThread() has been called on it. If so, the Thread function returns (terminating itself).
- ReceiveSocksToWaitOn.Wait()s until there is data to be received on one or more of the specified SOCKETs
- For each ready socket, the function retrieves the sent Packet (using UDP-Networking::ReceivePacket()) into the local memory buffer. It then does a linear scan of all Events on the mutexed list of Events. If it finds an Event that has a NetMessage that has the same DocumentNum as the locally buffered Packet, it NetMessage.ConcatenatePacket()s the locally buffered Packet to that NetMessage. Otherwise, it creates a NetMessage, NetMessage.ConcatenatePacket()'s the locally buffered Packet to that NetMessage, creates an Event (by timestamping the Event and recording the IP Address of the sender of the locally buffered Packet) and pushes the Event onto the mutexed list of Events.

Concerns The Thread object currently does not have access to a timer object, so it has no way of timestamping the Events that it creates. It must somehow be passed a reference to a timer object.

4.2.3 Server

The Server is a subclass of GameNetworking. It is to be instantiated whenever the local game is to act in a Server capacity, and destroyed whenever the local game should not behave like a Server.

Dependencies GameNetworking

Patterns To Be Applied Singleton

Implementation The Server privately encapsulates the number of milliseconds it has been since the last time a SERVER_UPDATE NetMessage has been GameNetworking::SendNetMessage()'ed to all the RemoteMachines.

Server encapsulates AcceptClient(), a public method that takes the IP Address of a Client and the Port at which that Client is receiving data at. First, it UDPNetworking::CreateSocket()'s a new SOCKET, and ReceiveSocksToWaitOn::AddSocket()s this SOCKET. AcceptClient() then adds the specified Client to the list of RemoteMachines, and NotifySelf()'s to ensure the Thread is responsive to the new Client's messages right away.

DoNetworking() takes no arguments. The method GameNetworking::ProcessEvents()'s, and GameNetworking::DetectRemoteMachineDelinquency()'s. It then updates timeSinceLastServerUpdated, and, if it's that time again, collaborates with the MessageHandler to SendNetMessage() a SERVER_UPDATE.

Server's constructor sets timeSinceLastServerUpdate = 0.

Server's destructor BootRemoteMachine()'s any RemoteMachines that are in contact.

```
class Server:GameNetworking
{
private:
    //timeSinceLastServerUpdate
public:
    //AcceptClient()
    //DoNetworking()
    //Server()
    //~Server()
};
```

Concerns None

4.2.4 Client

The Client is a subclass of GameNetworking. It is to be instantiated whenever the local game is to act in a Client capacity, and destroyed whenever the local game should not behave like a Client.

Dependencies
GameNetworking

Patterns To Be Applied
Singleton

Implementation
Client privately encapsulates a boolean value that describes whether or not it has successfully established contact with a Server. Connected() is a public method, which returns this value to the user.

Client's constructor takes the Server's IP Address and Port as arguments. It adds a RemoteMachine (filled out with the provided IP Address, Port and the GameNetworking::receiveSock) to the list of RemoteMachines.

DoNetworking() takes no arguments. It will GameNetworking::ProcessEvents(). If the Client is not Connected, then DoNetworking will GameNetworking::SendNetMessage()'s a HELLO NetMessage to the Server. If the Client is Connected, DoNetworking() will GameNetworking::DetectRemoteMachineDeliquency().

QuitServer() takes no arguments. It calls BootRemoteMachine(), passing the IP Address of the only RemoteMachine that should be on the list of the RemoteMachines – the server.

```
class Client:GameNetworking
{
private:
    bool connected;
public:
    Connected()
    DoNetworking()
    Client
    ~Client
};
```

The following are constant values defined in the GameNetworking namespace.

TIMEOUT is equal to the number of milliseconds before the MessageHandler is informed of a RemoteMachine's lack of communication.

PACKET_TIMEOUT is equal to the number of milliseconds that must pass before a partially constructed NetMessage (a NetMessage inside an Event [which is sitting on the Event list] still waiting to be NetMessage.ConcatenatePacket()'ed with one or more Packets) is discarded.

Concerns None

4.2.5 RemoteMachine

Overview
The RemoteMachine encapsulates all the information necessary to refer to a remotely connected game.

Dependencies
None

Patterns to Be Applied
None

Implementation RemoteMachine encapsulates the port number that one would GameNetworking::SendNetMessage() to when sending a NetMessage to this remote machine, the SOCKET that is earmarked to receive Packets from the remote machine, the time (in milliseconds) since this particular remote machine last communicated with the local machine, and the IP Address the remote machine resides at.

```
struct RemoteMachine
{
public:
    //PortToSendTo
    //SockeToReceiveFrom
    //TimeSinceLastUpdate
    //IPAddress
};
```

Concerns RemoteMachine acts as a collection of WinSock 2.2 datatypes. As such, it makes no effort to abstract the WinSock library and hence should be considered strictly a low level object.

4.2.6 ReceiveSocksToWaitOn

ReceiveSocksToWaitOn abstracts the concept of waiting for data to be received by a set of SOCKETs. It allows the user to use a high level interface to specify which SOCKETs to wait on, actually wait for data to be sent to one or more of the SOCKETs, and then retrieve which SOCKETs have Packets ready to be received. Finally, it allows the user to simulate internet latency by allowing the

user to specify how often, and for how long, batches of Packets will be discarded rather than received.

Patterns To Be Applied
None

Implementation ReceiveSocksToWaitOn privately encapusulates a mutexed list of SOCKETs that are to be waited on (waitOn), and a list of SOCKETs that have data ready to be received (ready). It also privately contains a boolean value that defines whether or not Packet loss simulation is active, another boolean value that defines whether or not Packets should currently be discarded rather than received, the $PacketLossOccurrence_{base}$, $PacketLossOccurrence_{delta}$, $PacketLossLength_{base}$, $PacketLossLength_{delta}$ information, and a timer object to allow the Packet loss simulation to operate as specified. Finally, it privately encapsulates the fd_set structure that is used to coerce WinSock to actually wait on the sockets listed in WaitOn.

ReceiveSocksToWaitOn's constructor clears both lists, sets both boolean values to false and zeroes out all Packet loss information, thus establishing a "blank slate" invariant.

ReceiveSocksToWaitOn provides AddSocket(), RemoveSocket() and RemoveAllSocket() functions, that behave as the user would expect to specify which SOCKETs should be waited on (ie adds or removes SOCKETs from the SOCKET list waitOn).

ReceiveSocksToWaitOn provides a public Wait() method. Wait() will block control flow until one or more of the SOCKETs on the waitOn list have data ready to be received on them. It will then return, and the SOCKETs that have data ready to be received on them will be listed on the private ready list. It accomplishes this by first removing all SOCKETs from the ready list and FD_ZERO()'ing the fd_set. Next, it FD_SET()'s every SOCKET in the waitOn list into the fd_set, and calls select() on the fd_set with an infinite waiting period. When a Packet arrives on one or more of the SOCKETs listed in the fd_set, select() will return, having already replaced the SOCKETs in the fd_set with the subset of SOCKETs that have data waiting to be received on them. Wait() puts all these SOCKETs into the ready list (via FD_ISSSET() and the waitOn list) and then returns.

ReceiveSocksToWaitOn provides a public ConfigurePacketLoss() method, which takes the base amount of time to simulate a burst of packet loss ($PacketLossOccurrence_{base}$), the random delta amount of time to modify this base time by ($PacketLossOccurrence_{delta}$), the base amount of time a burst of packet loss lasts for ($PacketLossLength_{base}$), and the random delta amount of time to modify this base time by ($PacketLossLength_{delta}$). All arguments are measured in milliseconds. If this function is called, then every $PacketLossOccurrence_{base} \pm PacketLossOccurrence_(delta)$ milliseconds, for $PacketLossLength_{base} \pm PacketLossLength_{delta}$ milliseconds, ReceiveSocksToWaitOn will throw away every incoming Packet. This is intended to simulate the "batches" of lost packets that is the hallmark of Internet latency.

Thread is made a friend of this ReceiveSocksToWaitOn, so that ReceiveThread can access the private list ready, thereby finding out which SOCKETs are ready to have data received on them.

```
class ReceiveSocksToWaitOn
{
private:
    //mutexed list<SOCKET> waitOn;
    //list<SOCKET> ready;
    //fd_set set;
    //Packet Loss Simulation Information
public:
    //AddSocket
    //RemoveSocket
    //RemoveAllSockets
    //Wait
    //ReceiveSocksToWaitOn
    //~ReceiveSocksToWaitOn
    //friend ReceiveThread;
};
```

Concerns None

4.2.7 Packet

Encapsulates the payload of a low level UDP message.

Implementation A Packet is a buffer of memory no larger than MAX_PACKET_SIZE. (unsigned char *)s are typedef'd to be (Packet *)s, for purposes of readability and abstraction.

- The zeroth, first, second, and third bytes of a Packet will be its Document-Num. Each Packet in a given NetMessage will have the same Document-Num, thereby allowing the Packets (each of which will be sent separately) to be reconstructed into the same NetMessage on the receiving end.
- The fourth byte of a Packet will be its PageNum. This zero based integer defines the order the Packets of the NetMessage should be read from. The NetMessage's map of Packets uses the PageNum as its key.
- The fifth byte of a Packet will be the total number of Packets in the NetMessage. In other words, if this value is greater than 1, then there are sibling Packets in the NetMessage.

- The sixth, seventh and eighth bytes of a Packet will be the size, in bytes, of the rest of the Packet (including the Packet ID).
- The ninth and tenth bytes of a Packet will the Packet ID. This will correspond to the type of information the rest of the Packet is carrying (for example, FISH_MOVE, or SERVER_UPDATE).

Concerns
None

4.2.8 NetMessage

The NetMessage is the high level unit of data that is GameNetworking::SendMessage()'ed to remotely participating machines.

Implementation NetMessage privately encapsulates a map of (Packet *)s that use the PageNum of the Packet as the key. It also contains private data such as curPacket (the current Packet being written to), curByte (the next byte within the current Packet that is to be written to), totalPages (the total number of Packets in the NetMessage), and a boolean value describing whether or not the NetMessage has entered Read Mode.

NetMessage contains a private boost::singleton_pool. This object is used to allocate and deallocate blocks of memory – Packets – of size MAX_PACKET_SIZE.

NetMessage uses the private function NewPacket() to properly create a new Packet. The function allocates a new Packet from the boost::singleton_pool, increments totalPages and curPacket, sets curByte to be equal to the first byte after the header information, sets the PageNum field of the new Packet to be equal to curPacket, and adds the new (Packet *) to the map,

NetMessage publicly encapsulates a suite of Write functions – Write1Byte(), Write2Byte(), Write4Byte() and Write8Byte(). Each of these functions takes a pointer to a (void *), and, provided Read Mode has not been entered, updates the Size field of Packet number curPacket to reflect its new size, and places the argument into the Packet at position curByte if there is still enough space remaining in the Packet to do so. If there isn't enough space in the current Packet to place the argument, then the Write function NewPacket()'s, increments totalPages and curPacket, updates the Size field of Packet number curPacket and places the argument into the current Packet at position curByte. Lastly, the function returns true. If Read Mode has been entered, the Write functions return false, performing no operations whatsoever. In this way, the user can simply Write out, in a linear fashion, the data they wish to send in the NetMessage without being concerned with the allocation of Packets or how the Packets will be reconstructed by the receiver.

NetMessage publicly encapsulates a complementary suite of Read functions – Read1Byte(), Read2Byte(), Read4Byte(), and Read8Byte(). Each of these functions take no arguments, and, if Read Mode has not yet been entered, puts the NetMessage into Read Mode, zeroes out curPacket and sets curByte to the

first byte after the Packet header information. Next, assuming curByte is not equal to the Size of the last Packet, these functions return a (void *) to the object at curByte in Packet number curPacket, and updates the curByte value (possibly reaching the end of the current Packet, in which case curByte is set to the first byte after the header information and curPacket is incremented). Attempts to read beyond the last byte of the last Packet cause all the Read functions to return 0.

NetMessage provides the public method GetPacket(), which takes an integral argument and returns a (void *) to the requested Packet, or 0 if the requested Packet does not exist. GetNumPackets() returns the number of Packets in the NetMessage. These methods allow the Networking module to send a NetMessage off, Packet by Packet.

NetMessage provides the public function GetID(), which simply returns the Packet ID of the first Packet in the NetMessage (after all, all Packets in a NetMessage must have the same ID). SetID() takes an unsigned short argument, and sets every Packet's ID field to be equal to that argument. This allows the creator of a NetMessage to set the ID of the NetMessage (ie SERVER_UPDATE, or FISH_BITE) without worrying about the formatting of Packet header information.

NetMessage provides the public method ConcatenatePacket(), takes a (void *) argument, NewPacket()'s, and then memcpy()'s the data from the argument to the new Packet.

NetMessage provides the public method DeallocatePackets(), which deallocates every (Packet *) in the map back to the boost::singleton_pool. That NetMessage object should thenceforth not be used.

NetMessage's public constructor zeroes out curPacket, curByte, and totalPages.

NetMessage's public destructor does NOT deallocate the (Packet *) in the map. This allows relatively small NetMessage objects to be passed about without copying the (potentially large) amount of data contained in the Packets, but forces the Networking module, and any module that receives NetMessages (for example, the Message Handler and Tank Simulator) to remember to NetMessage::DeallocatePackets() after the NetMessage's information has been read.

```
class NetMessage
{
private:
    //map<Packet DocumentNum, Packet *>
    //unsigned int curPacket
    //unsigned int totalPages
    //unsigned int curByte
    //bool readMode
    //boost::singleton_pool
    //NewPacket()
public:
```

```
    //WriteNByte()   //1, 2, 4 and 8 Bytes
    //ReadNByte()    //1, 2, 4 and 8 Bytes
    //GetPacket()
    //GetNumPackets()
    //GetID()
    //SetID()
    //ConcatenatePacket()
    //DeallocatePackets()
    //NetMessage
    //~NetMessage

};
```

Concerns

The NetMessage is far from an ideal object, as evidenced by the following concerns. It is highly recommended anyone making direct use of the NetMessage class read the following.

This class relies on the boost::singleton_pool object, which the team has no experience working with. If, for whatever reason, this object cannot be used, a memory manager that provides similar functionality (fast, non-fragmenting allocation of blocks of memory of a preset size) must be found or written.

NetMessage is intended to always be passed by value. This means that failure to NetMessage::DeallocatePackets() will cause memory leaks. This is unfortunate, especially for the modules that need to remember to, after reading all of the information from a NetMessage, perform the deallocation.

NetMessage is fragile in the sense that a Read or Write operation can never be taken back. Moreover, after a Read operation is enacted, attempts to perform Write operations will fail, with no effect.

NetMessage provides a relatively wide interface to all objects that need to use it. This is error-prone, as it is possible for objects who should simply pass the NetMessage along to inadvertently alter its data. Bugs such as this would be very difficult to detect.

4.2.9 Event

The Event simply encapsulates a NetMessage with the IP Address of the sender, and the time at which the NetMessage was received.

Implementation

An Event privately encapsulates a static timer object – which allows it to timestamp itself in its constructor.

An Event consists of the IP Address of the sender, followed by the time the Event was constructed, followed by the NetMessage itself. This data is public for easy, low level access.

Event's constructor accepts the IP Address of the sender and the NetMessage (passed by value) that was sent. It timestamps itself after initializing its members with the arguments passed.

```
class Event
{
private:
    //static timer object
public:
    //IP Address of the sender
    //Time Event was constructed
    //NetMessage
    //Event()
};
```

Concerns None

5 AI System

There are two functionally independent sections to the AI system. The first is the decision making process for objects within the game called the AI module. The second section is the genetic algorithm that governs the evolution of species in the game. In addition there will be a governing class known as the Brain Manager that encapsulates these two sections and provides other functionality for managing the game AI.

The AI module involves the processing of information to decide upon a course of action for a game object to take. These decisions will take the form of actions from a list of possible actions such as MOVETO and BITE. The data comes from other objects in the game object's sensor range and memory AI as well as internal, external and game states.

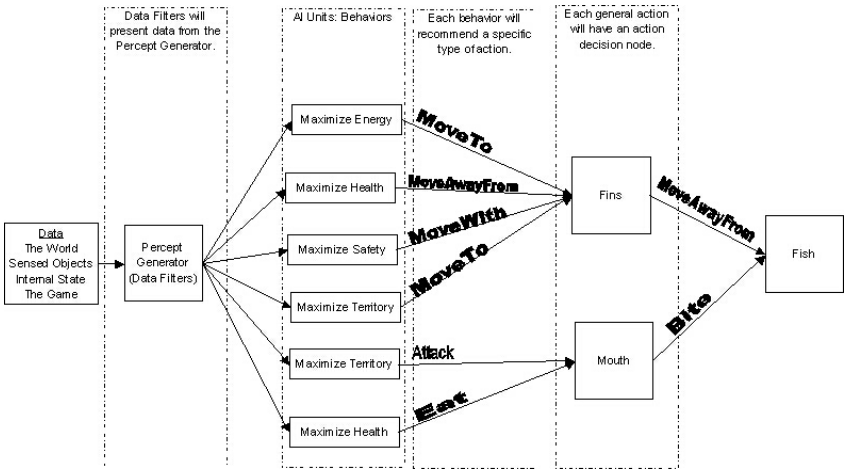
The genetic algorithm is the section that evolves the AI modules of future game objects by using crossover and mutation techniques.

5.1 AI Module

The AI module has two separate subsections. The first is percept generation that generates identifier-value pairs called attributes that are usable by the decision-making process. This provides an interface between the decision-making processes and the game world. The second section is decision-making that processes the percepts generated by the percept generator and outputs a list of actions to be interpreted by the game object.

The AI module will be applied to each object within sensor range of the game object and an action will be produced for each object. These actions will be evaluated and a configurable number of the most important actions will be selected and sent to the game object to be interpreted. If there are no objects within sensor range, the AI module will be run once and the actions generated will be sent to the game object for interpretation.

The AI module will also query the fish to determine which actions have been performed so it can notify the appropriate AI unit to save a training example.



5.1.1 Percept Generation

The percept generator is the interface between the game world and the decision-making process. It contains a number of data filters that query objects within the game to gather data. These data filters produce the attribute-value pairs that are used by the decision-making process. Fitness functions will function in the same way as data filters to evaluate the fitness of the fish.

The percept generator will reset the data filters at the end of processing and notify the data filters at the end of each pass of the AI module.

Percept Generator Data The following outline lists data available. Note that data is not limited to this list and additions may be made during implementation. Also note that if something is listed as an example it does not suggest that it is part of the game. It is only there to illustrate a point.

- Objects in sensor range
  - Determined through collision detection with the sensory volume.
  - List of handles to objects are generated
  - Examples
    - Other Fish
- Internal state
  - Physical attributes
    - Data available through internal state variables
    - Examples:
      - Health
      - Energy Level
  - Statistical attributes

        * Data available through statistics variables
        * Examples
            · Kill rate
            · Being attacked rate

- External state
    - Voxel attributes
        * Data available by querying the terrain object
        * Examples:
            · Plankton density
            · Current direction
    - Special locations
        * Collision detection with location objects
        * List of handles to special location objects generated
        * Examples:
            · Location objects
            · Havens

- Game state
    - Data available by querying the game
    - Examples:
        * Score

Data Filters Data filters are objects that the percept generator maintains to present data in a meaningful manner to the decision-making process. They do this by generating attribute objects that are identifier-value pairs. Attributes are used by the decision-making process to evaluate the game object's situation and determine actions for the game object to take.

Unless otherwise specified, data filters will only do their computations once when they are first invoked. The result will be stored and made available if the data filter is invoked again. After the AI module finishes processing, the data filters will be reset to ensure that values are re-calculated the next time the AI module is invoked. Some AI filters will require being reset with each pass of the AI module. They may do this when the percept generator notifies them to do so.

Some data filters may need to be trained. They will accomplish this by querying the fish for information to create training examples and will be allowed to train with the AI units. It may be decided to restrict the AI technique used for memory AI to an AI type that can train as it goes (such as Bayes Classifiers) in order to eliminate the need to have a dedicated training time.

One key feature of data filters is that they allow new methods of presenting data to the decision-making module without having to change any other part

of the AI module. If new data sources are added to the percept generator, new data filters can be included without old ones being changed. This prevents other portions of the AI module from going out of date when such an update occurs.

Fitness Functions Fitness functions provide a means for AI units and Action Decision Node's to train by providing feedback to the various machine learning techniques as to how they are doing. There are different types of fitness functions for the AI units and the Action Decision Nodes.

For AI units, fitness functions should be written such that they consider a single aspect of the game object to maximize such as energy level. For ADN's fitness functions are intended to represent higher goals. Some examples are aggressiveness, social ability and longevity.

Fitness functions work in almost the same way as data filters. They evaluate data available to the percept generator and produce an output based on this data. The difference is that fitness functions output either a positive or negative as well as a priority that is a real number in the range 0-1 intended to assign an importance value to the action specified by an AI Unit. The AI technique may choose whether to use the fitness, the response or both for training.

Special Note about Data Filters and Fitness Functions It is impossible to design a general AI system that can account for every detail in a specific implementation of the system. The data filters and fitness functions are intended to provide a means for the programmers and designers of a specific implementation to tailor the system to their needs. This leaves the onus of responsibility of breaking the world down to a form that can be interpreted by the AI system to them. This can be accomplished through the creative construction of data filters and fitness functions since the other techniques employed by the AI system are otherwise generic.

5.1.2 Decision Making

The decision-making module queries the percept generator for attributes that it uses to evaluate the current situation of the game object. It then makes decisions based on these attributes. It does this in two stages. The first is comprised of a list of AI units that make decisions intended to optimize a specific aspect of the game object. The second stage is a layer of specialized AI units called Action Decision Nodes. Action Decision Nodes take outputs from the AI units and choose one. They make their choices in an attempt to optimize higher goals provided by fitness functions.

AI Units AI units are the base AI objects of the AI system. They take as inputs attributes generated by the data filters in the percept generator and produce an action that can be interpreted by a game object. Associated with the action generated by an AI unit is a value that is intended to signify the importance placed upon the AI unit's decision.

The AI units employ one of several types of machine learning techniques. These include but are not limited to Bayes Classifiers, Decision Trees and Backward Propagation Neural Networks. These are trained from experience by generating training examples based on their inputs and a corresponding fitness function. The fitness functions associated with AI units will correspond to specific aspects of the game object (e.g. energy level).

Upon initial creation, the data filters and fitness function used by the AI unit are randomly chosen. This information along with the internal configuration specific to the AI technique will be able to be saved to a file for future use.

Action Decision Nodes Action Decision Nodes (ADN's) are a specialized form of AI Unit that produces as an output any of its inputs. They take as inputs the action-importance pairs generated by the AI Units.

There is one ADN for each type of independent action that an object can make. For example, a fish may be able to move and eat but a barnacle may only be able to eat, therefore the fish will have two ADN's but the barnacle will only have one. Likewise, the AI units connected associated with an ADN will only be able to specify actions that fall under the category of action that the ADN can specify.

ADN's are evaluated by fitness functions in the same manner as AI units, however the fitness functions associated with an ADN should reflect a higher goal than a fitness function associated with an AI unit.

5.1.3 Genetic Algorithm

Genetic algorithms are a form of machine learning where a population of potential solutions to a problem is evaluated and evolved in order to produce more fit solutions and eliminate less fit solutions. The result is a solution or a set of solutions to a problem.

1. Initialize a population of hypotheses
2. Evaluate the hypothesis
3. Select the most fit hypotheses
    (a) If a hypothesis provides a satisfactory solution to the problem, select that hypothesis and stop.
4. Remove the least fit hypotheses
5. Generate a new generation of hypotheses to add to the population using crossover techniques.
6. Add variance to the new population using mutation techniques
7. Repeat the process

5.2 Genetic Algorithm Module

Many of the steps involved with a genetic algorithm are taken care of as part of the nature of the game. Initialization of a population of hypotheses occurs when the initial AI modules are created for the initial population of fish. Fish are removed by their inability to survive in the game environment. The game will run as long as the user wishes to run it, so there is no end condition, thereby allowing the user to evolve the system as long as they wish to do so.

The portion of the genetic algorithm that will require direct control from the genetic algorithm module (GAM) is checking for "genetic" compatibility, crossover, mutation and generation of new populations.

When a fish decides it is time to "mate" it will put itself on the GAM" 's mating list. The GAM will then evaluate the fish and try to find a match for it from other fish on the mating list. If a match is found, it will then apply crossover and mutation techniques to generate offspring from the two fish. If no compatible match is found within a limited amount of time, the GAM will then create offspring from the single parent using mutation.

Evaluation of Fish Fish will be evaluated based on their ability to meet preset higher goals defined by the higher goal fitness functions used by the Action Decision Nodes. Every higher goal fitness function will be applied to the fish. The results of these will be compared to other fish in the mating list to determine which other fish match up the best. If a compatible mate is found, the two fish will be selected to reproduce.

Mutation Technique Mutation will simply involve the resetting of at least one AI unit in the fish's Decision-Making module.

Crossover Technique When two compatible fish are selected to mate, a random number of AI units will be selected from one parent and copied into the offspring. The difference will be copied from the other parent. Regardless of what AI units were copied, the new offspring will have new Action Decision Nodes.

5.3 Brain Manager

The brain manager will be the overarching class that will encapsulate the AI system. It is responsible for the creation, deletion and management of AI modules. It will maintain a list of fish and their reaction times along with their associated AI modules. Each game cycle it will determine which fish need to "think" and invoke that fish's AI module. It will then use the message handler to deliver the decisions to that fish.

In addition to managing the fish thinking, the brain manager will manage the genetic algorithm by calling the genetic algorithm's generateOffspring method and then managing a list of "fish to create" by sending "create fish" messages

to the tank. This is necessary since the tank may not always be ready to create a new fish every time a new one is ready to be created.

The user will be able to add fish randomly to the tank. The brain manager will handle this by keeping a count of the number of random fish to add. It will query the tank for a body and assign a brain to the body. It will continue to do this until the tank reports that no more bodies are available or until the random fish count has reached zero.

The brain manager will also query the tank to determine which fish are dead. If a fish is dead, its brain will be deleted and then the brain manager will send a message to the tank to delete the fish.

5.4 Implementation Design

The implementation design is intended to provide an outline of the core functionality of required classes and how they relate to each other. There will undoubtedly be a need for helper functions, accessor functions and other members, methods and operations that are assist the core design. These are implied and not included so as not to distract the reader from the focus of this part of the document.

5.4.1 Attribute

An attribute is a single piece of data. It consists of a type and a value. The AI units will make decisions based on attributes.

Attribute class

Members:

- ID
  - Identification of attribute
- value
  - Number used by AI unit to classify attribute
  - Used to create continuous valued attributes
  - Can be used to create Boolean valued attributes by restricting to 0 or 1

5.4.2 Actions

Actions are specified by the AI units and the Action Decision Nodes. They consist of identifier-value pairs where the value is the importance/magnitude of the action. They also contain an optional game object ID. This ID, if specified, will be set to the currently evaluated game object by the AI module.

Action class derived from Attribute class

Additional Members:

- ObjectID
  - ID of currently evaluated game object
- AIUnitID
  - ID of AI unit whose action was selected

5.4.3 Fitness Functions

Fitness functions work similar to data filters. They are used by an AI Unit so that the AI unit will know how well its doing. The fitness will be given as a number between 0 and 1.

Fitness Function base class

Members:

- dataProcessed:
  - Boolean flag
  - Indicates that the data filter has already processed the data
- Fitness:
  - Fitness generated by the fitness function
- Response:
  - Boolean flag that tells the AI Unit whether it is doing good or not
- Threshold:
  - Value that determines at what level of fitness produces a positive or negative response
- Priority:
  - Value that determines what importance an associated action is
- ID:
  - An id number used to identify the fitness function
- MessageHandler:
  - Reference to the message handler
  - Used to query the game for data

Methods:

- evaluateFitness:
  - Virtual function to be overridden by derived classes

    - Queries necessary data from the game
    - Performs necessary calculations with the data
    - Sets the fitness, priority and response
    - Returns response
- getPriority:
    - Returns the priority
- getFitness:
    - Returns the fitness
- getResponse:
    - Returns the response
- Clear:
    - Clears the fitness
    - Clears the priority

5.4.4 Percept Generator

The percept generator contains the data filters. A request is made for a type of data by giving the percept generator a data filter ID. The data filter is then used to generate an attribute and the results are passed on to the requestor.

Percept Generator base class

Members:

- dataFilters:
    - List of data filters
- sensoryData:
    - IDs to various game entities within the fish's sensor range

Methods:

- requestData:
    - Takes a Data Filter ID
    - Returns an attribute generated by that data filter
- reset:
    - Purges this rounds sensory data
    - Calls each data filter's reset method
- EndPass:
    - Informs data filters that the current AI module pass has ended

5.4.5 AI Units

AI Units are the core decision-making modules in the AI system. They take a list of attributes as input and produce a recommended action as an output. This is done using different AI techniques that will be implemented in derived classes.

AI Unit base class

Members:

- fitnessFunction:
  - Used to determine the fitness of an AI unit
- attributeList:
  - List of current attributes the AI Unit uses as input
- dataFilterIDs:
  - List of data filter IDs used to request information from the percept generator
- trainingExamples:
  - List of training examples
  - Limited size: FIFO overflow
- savedAttributeList:
  - If action for this AI unit is chosen the attributes are saved
  - Used to create a training example

Methods:

- determineAction:
  - Virtual function to be overridden by derived classes
  - Analyzes the attribute list and produces an Action
  - Gets the priority from the fitness function
- getAttributeList:
  - Calls upon the percept generator to generate a list of attributes
- Train:
  - Override-able function used by the various AI techniques to train
  - Trains the AI unit based on the trainingExamples list
- saveTrainingExample:
  - Calls the fitness function to get the current fitness of the AI unit
  - Saves the fitness along with the list of attributes from the saved attribute list as a training example in the trainingExamples list
- saveAttributes:

  - If the action from this AI unit is selected by the ADN this function will be called to save the attributes into the saved attribute list
- saveExamplesToFile:
  - Saves a set of training examples to a file
  - For space considerations, this may be cut in future design cycles
- loadExamplesFromFile:
  - Loads a set of training examples from a file
  - If saveExamples is cut this will be cut as well
- Save:
  - Saves the AI unit's configuration to a file
  - Saves the internal AI structure to a file. (e.g. neural network configuration and weights)
- Load:
  - Loads the AI unit's configuration from a file
  - Loads a saved AI structure from a file

5.4.6 Action Decision Nodes

Action decision nodes are a type of AI unit that take an action, chooses one of them and returns it. Each action decision node will be associated with a type of action that a game object is capable of that is independent with other actions.

Because action decision nodes are derived from AI units and actions are derived from attributes, action decision nodes will be able to use the same type of AI processing that is used by the AI units.

Action Decision Node class

Although ADN's are functionally the same as AI units they will be a separate derived class to provide type-safety as well as allow for changes that may be necessary in the future.

5.4.7 AI Module

The AI module will bundle up the system. It will have a percept generator, a list of AI Units, and a set of action decision nodes. It will take the data gathered from the sensors and output actions for the game creature to take. The game object will be responsible for implementing the actions, as it seems fit.

AI Module class

Members:

- Percept Generator

- AI Units:
  - List of AI Units
- decisionNodes:
  - List of decisionNodes

Methods:

- getSensorData
  - Gets sensor data for the percept generator
- generateDecisions:
  - Runs for each object in sensor range:
    - Queries the fish for a list of actions completed
    - If an action has been completed, it will tell that AI unit to save a training example
    - Calls each AI unit's determineAction function
    - Calls each action decision node's determineAction function
    - Generates a list of actions
  - Returns list of actions
- getPerformedActions:
  - Gets actions performed by the fish
  - Determines which AI unit performed the action and tells that AI unit to save a training example
- Create:
  - Purges any AI Units and ADN's currently in the AI module
  - Creates AI Units
  - Connects AI Units to Action Decision Nodes
- Train:
  - Trains any AI units that are ready to do so
- Reset:
  - Resets all AI Units
  - Resets all Action Decision Nodes
- Save:
  - Saves the AI module to a file
- Load:
  - Loads the AI module from a file
- GetAIUnit:
  - Gets a specified AI unit
- PutAIUnit:
  - Puts a given AI unit into the AI module
- Mutate:
  - Resets a random AI unit

5.4.8 Genetic Algorithm Module

The genetic algorithm module (GAM) will be a system wide component of the AI system. When a fish is ready to mate it puts itself on the GAM's mating list. The GAM will then find a match for the fish and create offspring. If there are no other eligible fish the GAM will produce offspring using mutation.

Members:

- Mating list:
  - List of game objects and a time stamp

Methods:

- addToMatingList:
  - Adds a game object to the mating list
- GenerateOffspring:
  - Called by the game to generate off spring from the mating list
  - Searches mating list for compatible mates
  - Applies crossover or mutation to create offspring
- Crossover:
  - Applies crossover techniques to two compatible game objects
  - Determines whether or not to apply mutation to the offspring
- Mutation:
  - Applies mutation techniques to a single game object to create offspring

5.4.9 Brain Manager

The brain encapsulates the system and provides an access point for the rest of the game to interface with the AI system. It will manage creation, deletion and management of all AI modules.

Members:

- Brains:
  - List of active AI modules in the system
- NewBrains:
  - List of newborn brains waiting for a body
- GeneticAlgorithm:
  - A genetic algorithm module

Methods:

- CreateFishBrain:

  - Creates a brain given a brain configuration
  - Tells tank to create a fish body
- SuppressBrain:
  - Suppresses a brain given an object ID
- SuppressBrainWithAction:
  - Suppresses a brain given an object ID and an actionID
  - ActionID used to train AI unit
- AddToMatingList:
  - Adds a given fish to the mating list
- CreateRandomFishBrain:
  - Creates a brain for a fish with random hookups
  - Tells tank to create a body
- Logic:
  - Checks list to determine whether or not a fish is ready to think
  - Activates the AI module for a fish ready to think
  - Sends decisions out to fish that have thought
  - Checks the dead fish list in the tank to determine which brains to delete
  - Deletes brains of fish that are dead and not on the mating list
  - Inactivates brains of fish that are dead and on the mating list
- Breed:
  - Plays match maker for fish on mating list
  - Creates new brains for the fish
  - Requests a body from the tank
  - If no body is available, adds brain to NewBrains list until a body is available
- Train:
  - Allows AI modules to train

5.5 AI Techniques

The game will employ a number of machine learning techniques. These include Bayes classifiers, decision trees and backward propagation neural networks. These techniques are similar but have different strengths and weaknesses. The AI system is flexible enough to allow other techniques as well as variations of the techniques that are listed in the following sections.

5.5.1 Naive Bayes Classifiers

The Naive Bayes Classifier is based on Bayes theorem. It is a probabilistic method of machine learning that does not require a search through the space of possible hypothesis. Instead, decisions are made based on frequencies of occurrence in the training data. It has been found to have performance comparable to neural networks or decision trees.

Note: This is a distillation of the material presented by Tom Mitchell in his book "Machine Learning." For a more detailed discussion of Bayesian learning please refer to his book.

Bayes Theorem The underlying principle of the Bayesian Learning is Bayes Theorem:

$$P(h/D) = \frac{P(D/h)P(h)}{P(D)}$$

This is to say that the Probability (P) of a hypothesis (h) given the data (D) is equal to the probability of the data given the hypothesis time the probability of the hypothesis divided by the probability of the data.

***Maximum A Posteriori* (MAP) hypothesis** The MAP hypothesis is the most probable hypothesis (h) from some set of hypotheses H ($h \in H$). The MAP hypothesis can be found using Bayes Theorem:

$$\begin{aligned} h_{\text{MAP}} &= \arg\max P(h/D) \\ h_{\text{MAP}} &= \frac{P(D/h)P(h)}{P(D)} \\ h_{\text{MAP}} &= P(D/h)P(h) \end{aligned}$$

$P(D)$ is dropped in the last term because it is a constant independent of h.

The above concepts are used in machine learning by considering D to be a set of training examples and h as a target value or function from a set of possible target values (H).

Naive Bayes Classifier From a set of training examples a Bayesian approach to learning would be to determine the probability of each attribute based on each potential outcome. Then when given a new instance to classify, use these probabilities in conjunction with Bayes Theorem to classify the instance.

$$v_{\text{MAP}} = \arg\max P(v_j|a_1, a_2, \cdots, a_n)$$

From Bayes Theorem this can be rewritten as:

$$v_{\text{MAP}} = \frac{P(a_1, a_2, \cdots, a_n|v_j)P(v_j)}{P(a_1, a_2, \cdots, a_n)}$$
$$v_{\text{MAP}} = P(a_1, a_2, \cdots, a_n|v_j)P(v_j)$$

The Naive Bayes Classifier is based on assuming that the attributes are conditionally independent given the target value (which is a very naive assumption). This allows the probability of observing the conjunction of attributes to be the product of the probabilities for each individual attribute.

$$P(a_1, a_2, \cdots, a_n|v_j) = \prod_i P(a_i|v_j)$$

Substituting this into the previous equation we get the Naive Bayes Classifier:

$$v_{\text{NB}} = \arg\max \prod_i P(a_i|v_j)$$

m-Estimate of Probability It should be noted that there are several ways to calculate the probabilities used in the Naive Bayes Classifier. The simplest method is the percentage of occurrences to the total number of instances for a particular outcome. However, this method can cause problems when the number of instances of an attribute is small relative to the whole. This is readily apparent since the calculated target value involves a product of probabilities and if just one of those were at or near zero it would cancel out all others. To overcome this, Naive Bayes Classifiers often use the m-estimate of probabilities as given here:

$$\frac{n_c + mp}{n + m}$$

Where n_c is the number of occurrences of a certain attribute given a certain outcome, n is the number of occurrences of that outcome, p is the prior probability estimate and m is a constant known as the equivalent sample size.

The variables n_c and n are easy to understand since the ratio $\frac{n_c}{n}$ is the simple percentage previously mentioned. The variable p is the last known estimate of the attribute. If there is no known last estimate, all values of an attribute can be considered equally probable and p can be set to k^{-1} for k possible values of an attribute. The variable m can be taken as the total number of possible values for the attribute. It should be noted that if all priors (p) are considered equally probable then the above equation simplifies to:

$$\frac{n_c + 1}{n + m}$$

Algorithm for Training

- Given a list of training examples
  - Determine the probabilities of each outcome
    - Example: Predator = 0.20
    - Example: Prey = 0.80
  - Determine the probabilities of each attribute for each outcome
    - Example:
      - Sharp Teeth given Predator = .90
      - Sharp Teeth given Prey = 0.10
    - Example:
      - Small Size given Predator = 0.30
      - Small Size given Prey = 0.70

Algorithm for Decision Making

- For each attribute multiply the appropriate probability for each possible outcome
  - Example: Fish is small with sharp teeth
    - Predator? $= P(\text{Predator}) \cdot P(\text{sharp teeth}|\text{predator}) \cdot P(\text{small}|\text{predator}) = 0.20 \cdot 0.90 \cdot 0.30 = 0.054$
    - Prey? $= P(\text{Prey}) \cdot P(\text{sharp teeth}|\text{prey}) \cdot P(\text{small}|\text{prey}) = 0.80 \cdot 0.10 \cdot 0.70 = 0.056$
    - Choose Prey since $0.056 > 0.054$
- Determine the conditional probability by normalizing the results
  - Example:
    - Probability that fish is prey is: $0.056/(0.054 + 0.056) = 0.51 =$ 51% chance that this fish is prey
  - Assign this value as the attribute importance for the decision outcome

Example Given the following attributes:

Attributes	Possible Values		
Size	(S)mall	(M)edium	(L)arge
Speed	(S)low	(A)verage	(F)ast
Teeth	(D)ull	(A)verage	(S)harp

Given the following training examples:

Example	Size	Speed	Teeth	Predator?
1	S	F	A	N
2	M	F	S	Y
3	S	A	A	Y
4	L	S	D	N
5	M	A	A	N
6	M	F	D	N
7	S	S	S	N
8	L	S	A	N
9	L	A	D	N
10	S	F	A	Y

Break the table into two tables for each outcome:

Predator = No Total examples = 7

Size	Small = 2	Medium = 2	Large = 3
Speed	Slow = 3	Average = 2	Fast = 2
Teeth	Dull = 3	Average = 3	Sharp = 1

Predator = Yes Total examples = 3

Size	Small = 2	Medium = 1	Large = 0
Speed	Slow = 0	Average = 1	Fast = 2
Teeth	Dull = 0	Average = 2	Sharp = 1

Determine the probabilities for each using the m-estimate:

Note because of the contrived nature of this example the m-estimate formula will be $\frac{n_c+1}{n+3}$ for each attribute. Not all attributes need to have the same number of potential values so this could be different for each attribute type.

Predator = No Probability = $\frac{7+1}{10+2} = .667$

Size	Small = $\frac{2+1}{7+3} = .3$	Medium = .3	Large = .4
Speed	Slow = .4	Average = .3	Fast = .3
Teeth	Dull = .4	Average = .4	Sharp = .2

Predator = Yes Probability = .333

Size	Small = .5	Medium = .33	Large = .166
Speed	Slow = .166	Average = .33	Fast = .5
Teeth	Dull = .166	Average = .5	Sharp = .33

Now we can classify new instances such as the following:
Fish A: Small, Average, Sharp
Probability of "Not Predator":

$P(\text{No}) \cdot P(\text{Small}|\text{NO}) \cdot P(\text{Average}|\text{NO}) \cdot P(\text{Sharp}|\text{NO}) = 0.667 \cdot 0.3 \cdot 0.3 \cdot 0.2 = 0.0120$

Probability of "Predator":

$P(\text{Yes}) \cdot P(\text{Small}|\text{Yes}) \cdot P(\text{Average}|\text{Yes}) \cdot P(\text{Sharp}|\text{Yes}) = 0.33 \cdot 0.5 \cdot 0.33 \cdot 0.33 = 0.0180$

Choose "yes". Calculate conditional probability: $0.018/(0.012 + 0.018) = 0.60$

Therefore the naive bayes classifier would classify this fish as a predator and assign the value of 0.6 to the return attribute (i.e. Predator with 60% certainty).
Fish B: Large, Fast, Average
Probability of "Not Predator":

$$P(\text{No}) \cdot P(\text{Large}|\text{NO}) \cdot P(\text{Fast}|\text{NO}) \cdot P(\text{Average}|\text{NO}) = 0.667 \cdot 0.4 \cdot 0.3 \cdot 0.4 = 0.0320$$

Probability of "Predator":

$$P(\text{Yes}) \cdot P(\text{Large}|\text{Yes}) \cdot P(\text{Fast}|\text{Yes}) \cdot P(\text{Average}|\text{Yes}) = 0.33 \cdot 0.166 \cdot 0.5 \cdot 0.5 = 0.0140$$

Choose "no". Conditional probability $= 0.032/(0.032 + 0.014) = 0.70$
Therefore the naive bayes classifier would classify this fish as not a predator and assign the value of 0.7 to the return attribute (i.e. Not a predator with 70% certainty).

5.5.2 Implementation

The Bayes Classifier specific implementation details are as follows.

The Bayes Classifier privately encapsulates the total number of stored training examples, the total number of Choices that the BayesClassifier has available to it, and a container of Choices.

```
class BayesClassifier
{
private:
    unsigned int numTrainingExamples;
    unsigned int numChoices;
    Choice choice[numChoices];
public:
    BayesClassifier();

    MakeDecision();
    Train();

    AddChoice();
    RemoveChoice();
    AddDataFilterKindInChoice();
    RemoveDataFilterKindInChoice();
    AddCategoryToDataFilterKindInChoice();
    RemoveCategoryFromDataFilterKindInChoice();
};

MakeDecision(DataFilters)//current situation
{
    double choiceProbability[numChoices];
    for(int i=0; i < numChoices; ++i)
    {
```

```
        choiceProbability[i] =  (choice[i].GetNumOccurrencesOfChoice() + 1) /
                                (numTrainingExamples + numChoices);

        for(int j=0; j < DataFilters.Num; ++j)//every datafilter in situation
        {
            choiceProbability[i] *=
            (choice[i].numOccurrenceOfAttribute(    Datafilter[j].ID,
                                                    DataFilter[j].value)
                                                    + 1)
            /   (choice[i].GetNumOccurrencesOfChoice() +
                choice[i].GetNumCategoriesOfDataFilterKind(Datafilter[j].ID));
        }
    }
    choice = MaxOfAll(choiceProbability);
    certainty = choiceProbability[choice] / (SumOf(other choiceProbability));

    return;
}

Train(ChoiceID, AttributeList)
{
    ++numTrainingExamples;
    choice[ChoiceID].IncrementOccurrenceOfChoice();
    for(i=0; i < AttributeList.size(); ++i)
    {
        choice[ChoiceID].IncrementOccurrenceOfAttribute(AttributeList[i].ID,
                                                        AttributeList[i].value);
    }
}
```

AddChoice() takes an ID (the ChoiceID – pass the ID of the Action this Choice corresponds to), and a reference to a container of IDs (the IDs of some DataFilterKinds – pass the IDs of the Attributes that are to be considered for this Choice). It appends this choice to the container of Choices in the BayesClassifier.

RemoveChoice() takes an ID (the ChoiceID – pass the ID of the Action this Choice corresponds to). It removes the specified Choice from the BayesClassifier. If the specified Choice is not found, a Nexception is thrown.

BayesClassifier::AddDataFilterKindInChoice() takes an ID (the ChoiceID – pass the ID of the Action this Choice corresponds to), another ID (the DataFilterKindID – pass the ID of the Attribute the Choice is to start considering) and a reference to a container of pairs (where the pair.first() is the lowerBound of a given Category, and pair.second() is the upperBound of a given Category). It forwards all arguments but the first to choice[ChoiceID].AddDataFilterKind().

Choice::AddDataFilterKind(), creates a new DataFilterKind with the specified ID and the specified Categories (one Category for each pair, where pair.first() is the lower boundary and pair.second() is the upper boundary).

BayesClassifier::RemoveDataFilterKindInChoice() takes an ID (the ChoiceID – pass the ID of the Action that corresponds to the Choice that should stop considering the specified DataFilterKind), and another ID (the DataFilterKindID – pass the ID of the Attribute the Choice is to stop considering). If ChoiceID is not found, a Nexception is thrown. It forwards DataFilterKindID to choice[ChoiceID].RemoveDataFilterKind().

Choice::RemoveDataFilterKind, removes the specified DataFilterKind. If the specified DataFilterKindID is not found, a Nexception is thrown.

BayesClassifier::AddCategoryToDataFilterKindInChoice() takes an ID (the ChoiceID – pass the ID of the Action the Choice that corresponds to), and another ID (the DataFilterKindID – pass the ID of the Attribute that corresponds to the DataFilterKind to add a Category to) and two floating point values – the lower boundary and upper boundary, respectively, of the Category to be added. It forwards every argument (except the first) to choice[ChoiceID].AddCategoryToDataFilterKind().

Choice::AddCategoryToDataFilterKind() forwards all but the first argument to dataFilterKind[DataFilterKindID].AddCategory().

DataFilterKind::AddCategory() adds the specified Category to the container category. It then calls DataFilterKind::Validate() to ensure that the DataFilterKind's category container is still valid.

BayesClassifier::RemoveCategoryFromDataFilterKindInChoice() takes an ID (the ChoiceID – pass the ID of the Action the Choice that corresponds to), and another ID (the DataFilterKindID – pass the ID of the Attribute that corresponds to the DataFilterKind to remove a Category from) and two floating point values – the lower boundary and upper boundary, respectively, of the Category to be removed. It forwards every argument (except the first) to choice[ChoiceID].RemoveCategoryFromDataFilterKind().

Choice::RemoveCategoryFromDataFilterKind() forwards all but the first argument to dataFilterKind[DataFilterKindID].RemoveCategory().

DataFilterKind::RemoveCategory() removes the specified Category (that is, the Category that matches the lower bound and upper bound arguments) from the container category. It then calls DataFilterKind::Validate() to ensure that the DataFilterKind's category container is still valid.

Choice encapsulates an ID (which is equal to the corresponding ActionID), the number of times this Choice has occurred in training examples, and a container of DataFilterKinds, which enumerates every possible Data Filter the Choice can be correlated with (in other words, if dataFilterKind contains only elements with IDs corresponding to Size and Teeth, then the BayesClassifier will only consider Size and Teeth when deciding whether or not to select this Choice).

```
class Choice
{
private:
    unsigned int ID;
    unsigned int numOccurrenceChoice;
    unsigned int numDataFilterKind;
```

```
    DataFilterKind dataFilterKind[numDataFilterKind];
public:
    Choice();

    GetID();
    GetNumOccurrencesOfChoice();
    GetNumCategoriesOfDataFilterKind();
    IncrementOccurrenceOfChoice();
    IncrementOccurrenceOfAttribute();

    AddDataFilterKind();
    RemoveDataFilterKind();
    AddCategoryToDataFilterKind();
    RemoveCategoryFromDataFilterKind();
};
```

GetID() takes no arguments. It returns ID.

GetNumOccurrencesOfChoice() takes no arguments. It returns numOccurrenceChoice.

IncrementOccurrenceOfChoice() takes no arguments. It increments numOccurrenceChoice.

IncrementOccurrenceOfAttribute() takes an (integral) DataFilterKindID and a (floating point) DataFilterKind value. It then forwards the second argument to dataFilter[DataFilterKindID].AddOccurrence(). If DataFilterKindID is not found, a Nexception is thrown.

DataFilterKind::AddOccurrence() finds the category that the floating point value falls into, and increments that category's numOccurrences value. If the floating point value is lower than the lowerBound of the category with the lowest valued range or higher than the upperBound of the category with the highest valued range, then the value is "clamped" to the lowest valued category or the highest valued category, respectively.

DataFilterKind encapsulates an ID (that is equal to the ID of the corresponding Attribute (Teeth, for example)), and the Categories that this DataFilterKind's floating point values have been broken into.

```
class DataFilterKind
{
private:
    unsigned int ID;
    unsigned int numCategories;
    Category category[numCategories];
    Validate();
public:
    DataFilterKind();
    GetID();
    GetNumOccurrencesOfValue();
    AddOccurrence();
```

```
    AddCategory();
    RemoveCategory();
};
```

GetID() takes no arguments and returns ID.

Validate() takes no arguments. It iterates over the container of Category's, and, if the Category's do not constitute a continuous range of values (that is, the first category's upperBound is equal to the second category's lowerBound, for every pair of Categories save the first and last), it throws a Nexception.

Category encapusulates the lowerBound and upperBound of the range of floating point values it encompasses, and the number of times this range of values has been encountered in past training examples (that is, if this Category resides in the "Size" DataFilterKind, and that DataFilterKind resides in the "Not Predator" choice, then numOccurrences is equal to the number of times an object whose Size fell into the range defined by lowerBound and UpperBound was classified as "Not a Predator").

```
struct Category
{
    double lowerBound;
    double upperBound;
    unsigned int numOccurrences;

    Category();
};
```

5.5.3 Decision Trees

Decision trees are a method of machine learning that classifies a system based on attributes of that system. The result is a tree structure that can be interpreted as if-then rules. This technique has interesting applications to AI in games since it can be used to generate AI code for game objects. Although Decision Trees have been studied by a number of people, the most notable is J. Ross Quinlan who developed ID3 and C4.5.

Limitations Decision trees rely on discrete valued inputs. There are methods for handling continuous valued inputs but these methods simply break the continuous values into thresholds thereby "digitizing" them. Binary decision trees are especially affected by this limitation and although there are methods of creating n-tree decision trees, the algorithms become more complex.

Another limitation is the inability of decision trees to grow without scrapping the old tree and generating a new one. There are methods which have been studied to avoid this such as Utgoff's ID5R. Due to our limited time and manpower, these techniques will only be considered upon completion of other team goals.

For a more detailed discussion of the limitations of decision trees please refer to Quinlan's book "C4.5 Programs for Machine Learning."

Entropy Entropy is the measure of the purity of a set of data. The higher the entropy the more impure is the data. The formula for entropy is as follows:

$$\text{Entropy} \qquad S = -p_+ \lg(p_+) - p_- \lg(p_-)$$

Where:

$$p_+ = \text{percentage of positive examples to the total}$$
$$p_- = \text{percentage of the negative examples to the total}$$

This equation gives numbers in the range of 0 to 1. If all attribute values are either positive or negative, the entropy will be 0. This shows that the data is very pure. If the attribute values are evenly split, the entropy will be 1.0 which is the maximum impurity of the data.

Example *Note: This example is taken from Machine Learning by Mitchell p. 56.* Suppose S is a collection of 14 examples of some boolean concept including 9 positive and 5 negative examples. Then the entropy of S relative to this boolean classification is:

$$\text{Entropy}(9+, 5-) = -(9/14)\lg(9/14) - (5/14)\lg(5/14) \approx 0.940$$

Information Gain Information gain is a measurement of the reduction in entropy of a system if a given attribute is chosen as a classification of the data. Information gain for a given attribute is given by the following equation:

$$\text{Gain}(S, A) = \text{Entropy} = \sum_{v \in \text{Values}(A)} \frac{S_v}{S} \text{Entropy}(S)$$

Where:

$$\text{Entropy}(S) = \text{overall entropy of the system}$$
$$S = \text{number of examples in the system}$$
$$S_v = \text{number of examples that have the value } v \text{ in } A$$

When building a tree, information gain is used to determine the attribute to choose for each node. The attribute with the highest information gain is chosen as the current node and all examples that match each value of an attribute are used as the examples to build each subtree.

Example *Note: This example is taken from Machine Learning by Mitchell p. 56.* From the previous example, consider an attribute called *Wind* with values *Weak* and *Strong*. There were 8 examples with strong wind and 6 examples with weak wind. From the days with strong wind there were 6 examples that registered as positive with the target attribute and 2 that were negative applying

the entropy equation for these eight examples yields an entropy value of 0.811. From the six days with weak wind there were 3 days with positive results and 3 days with negative results. From the entropy equation these six examples have an entropy value of 1.0. The information gain for the wind attribute is thus:

$$\text{Gain}(S, Wind) = 0.940 - (8/14)0.811 - (6/14)1.00 = 0.048.$$

Decision Tree Algorithm For the purposes of this project we will be using a binary decision tree with continuous attributes. Incorporating continuous attributes is a necessity for this project and it will require the additional steps of determining a threshold value. Keeping the tree binary will keep the implementation simpler than trying to develop an n-tree decision tree.

Building the Binary Decision Tree with Continuous Attributes The following outline is a high level algorithm for building a decision tree with continuous attributes.

- If all examples are positive with respect to the target attribute (fitness function) then label the current node as positive and return.
- If all examples are negative with respect to the target attribute (fitness function) then label the current node as negative and return.
- If there are no training examples then set the current node to a default value and return
  - Examples of things that can be done for a default
    - Set the current node to the most common value of the target
    - Choose positive for the right subtree and negative for the left subtree
      - For each threshold set
        - Determine the entropy
        - Count + and - examples
        - Apply entropy formula
      - Apply the gain formula
    - Randomly set the final node
- Determine the Current node
- Determine the information gain for each attribute
  - Determine the threshold value
    - Average of where the set transitions from + to - or - to +
    - Determine which threshold has the highest gain
    - Return the threshold with the highest gain

  - Return the gain of the highest threshold

- Choose the attribute with the highest gain as the current node
- Create the left subtee by calling "build tree" with a list of all training examples that have a value of the current attribute below the threshold for that attribute.
- Create the right subtree by calling "build tree" with a list of all training examples that have a value of the current attribute above the threshold for that attribute.

Traversing a Decision Tree The following outline is a high level description of how to traverse a binary decision tree with continuous attributes.

- If the current node is labeled as positive return a positive decision (likewise for negative decision)
- Check the attribute from the current set of inputs versus the current node
  - If it is less than the threshold, traverse the left subtree
  - If it is greater than the threshold, traverse the right subtree

5.5.4 Backward Propagation Neural Networks

Neural networks are probably the most talked about form of machine learning. The motivation behind the initial research on neural networks was based on modelling the behavior of neurons from nature. As a result neural networks are usually comprised of interconnected units that affect the other units that are connected to them. Much work has been done on this form of machine learning and there are many different flavors of neural networks. The type of neural network described here is known as "Backward Propagation".

Limitations Unlike Bayes classifiers and decision trees neural networks have no problem dealing with either real or discrete valued inputs and producing real, discrete or vector valued outputs. One limitation is that it is almost always impossible for humans to be able to understand or interpret the target function that the neural network produces. Another is that, although decision-making tends to be quick, training can take more time than is acceptable for real time learning. This may require that learning be done offline. However, the network can be trained using a "learn as you go" approach.

Sigmoid Units A common form of neural network is based on a unit known as a sigmoid unit. Sigmoid units essentially represent a linear equation with each input weighted by a value that is adjusted during training. The output is a differentiable non-linear function of its inputs. In this form of neural network,

sigmoid units are connected to form one or more layers to handle varying levels of complexity of the data.

The sigmoid unit stores a set of weights that are applied to its inputs. Often there is an initial weight (w_0) that is not associated with an input value. This is known as the bias since its affect on the outcome is constant regardless what the inputs are. The weights are initialized to random values and then adjusted through training.

The following represents sigmoid units:

$$o = \sigma(w_0 \cdot 1 + w_1 \cdot x_1 + \cdots + w_n \cdot x_n) = \sigma(\vec{w} \cdot \vec{x})$$
$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

Where:

o	=	the output of the sigmoid unit
$\vec{w}$	=	the vector of weights associate with each input
$\vec{x}$	=	the vector of sigmoid unit inputs
σ	=	the squashing function

The squashing function is so named because it maps a large input domain into the range of outputs between 0 and 1. As an alternative, tanh can be used.

Weight Training: Backward Propagation The weights used in the sigmoid units are trained based on training examples. There are two instances during which training can take place. One is to buffer training examples and then use them to train the neural network by running through the training algorithm making adjustments to the weights until the output of the neural network is sufficiently accurate in meeting a target value. The other is to train the network with each new instance provided that the output of the neural network can be evaluated with each new instance via a fitness function.

Training of weights is achieved by propagating the errors made by the network backward from the output to the hidden layers. The weights are adjusted to account for the errors.

The output errors are calculated by the following:

$$\delta_k \longleftarrow o_k(1 - o_k)(t_k - o_k)$$

Where:

δ_k	=	the error term for output unit k
o_k	=	the output from output unit k
t_k	=	the target value for the output (e.g. output from a fitness function)

The errors are propagated back through the neural network using the following:

$$\delta_{i,j} \longleftarrow o_{i,j}(1 - o_{i,j}) \sum_{k \in \text{outputs}} w_{i+1,j,k}\delta_{i+1,k}$$

Where:

$\delta_{i,j}$ = the error output from layer i, sigmoid unit j
$\delta_{i+1,j}$ = the error output from layer $i+1$, sigmoid unit j
$o_{i,j}$ = the output from layer i, sigmoid unit j
$w_{i+1,j,k}$ = the weight associated with the output of sigmoid unit i, layer j in sigmoid unit k from layer $i+1$

The weights are then updated using the following:

$$w_{i,j,k} \longleftarrow w_{i,j,k}\eta\delta i, jx_{i,j,k}$$

Where:
$w_{i,j,k}$ = the k^{th} weight from layer i, sigmoid unit j
η = the learning rate
$\delta_{i,j}$ = the j^{th} sigmoid unit from layer i
$x_{i,j,k}$ = the k^{th} input from layer i, sigmoid unit j
Note: the term $\eta\delta_{i,j}x_{i,j,k}$ is also referred to as $\Delta w_{i,j,k}$.

Creation of the Network The number of layers and sigmoid units per layer are determined. This may be user configurable however. Once set these do not change. The weights are then randomly set to small values (-0.5 to 0.5 are typical).

Training If training examples are available, the network may be trained using the training examples available until the network reaches an acceptable accuracy.

If training examples are not available or if it is desired to further train the network through experience, the network can be trained with each instance that it is given to classify provided that there exists a method of evaluation that can be used as a target value.

Training is done using the backward propagation method described above.

Decision Making Single output decisions are made from comparing the output of the final sigmoid unit with a threshold value. If the output is above the threshold value the result is positive otherwise it is negative.

Multiple output decisions are made by taking the output with the highest value as the decision.

5.6 References

Mitchell, Tom M. (1997). Machine Learning. Boston, MA: WCB/McGraw-Hill.
Quinlan, J. Ross (1993). C4.5 Programs for Machine Learning. San Mateo, CA: Morgan Kaufman Publishers Inc.

6 Tank Simulator

The tank simulator is primarily responsible for organizing the interactions between the fish, objects, and plankton.

6.1 Definition of Terms

Tank simulator client – an accessor object that allows objects in the simulation to interact with the tank simulator without allowing them to have a direct reference to the tank simulator itself.

6.2 Primary Interfaces

- The tank simulator requires no special initialization.
- A time step function will be provided, which will increment the simulation.
- A fish insertion function will generate a new, unformatted, fish body at (0,0,0); and may return an error value that will explain why the instantiation process failed.
- A fish formatting function that will accept a Key and a reference to a frozen fish; and may return an error value that signifies that the requested fish body does not exist.
- A function that will return a frozen fish when given a corresponding Key.
- A fish deletion function that will accept a Key to a specific fish body that will be removed from the simulation.
- Accessor functions to read and write to the mating list, add fish list, delete fish list, and random fish creation counter.
- A large collection of functions that can be used to gather information about objects in the simulation; each will be extremely granular in scope.
- The tank simulator will be capable of creating and processing both network and file mementos.

6.3 Dependencies

- The physics simulator.
- The graphics module and its graphics clients.
- The math library classes.
- The standard C++ library.

6.4 Internal Details

- Each tank simulator client will allow objects in the simulation to:
  - Deal damage to other objects in the simulation by passing in its Key and a damage quantity.
  - Place their Key on the mating list, which is periodically polled by the brain manager and used to generate genetically altered fish.
  - Place a frozen fish/brain pair onto an "add fish" list, which is periodically polled by the brain manager and used to generate precise fish in specific locations in the aquarium.
  - Place a Key onto a "delete fish" list, which is periodically polled by the brain manager and used to remove corresponding fish brains and bodies from the game.
  - Increment a random fish spawn counter, which is periodically polled by the brain manager and used to figure out how many new random fish need to be added to the tank.
  - Add any plankton quantity to a specific location in the tank.
  - Suck an amount of plankton from the environment based on the position, velocity, and suckability statistics of the fish that is sucking up the plankton.
  - Place fish on the "add fish" list by passing in a frozen fish/brain pair.
  - The tank simulator will use an STL map to quickly retrieve references to various objects when supplied with a Key.
  - The tank simulator will store all fish in an STL list.
  - The tank simulator will store all non-fish objects in an STL list.
  - The tank simulator will store plankton using voxels, in other words, a three dimensional array of floating point numbers.
  - The tank simulator will contain a physics simulator object.
- The tank simulator will perform the following during a time step call:
  - Set the game time step to a constant value if the global game state is OFFLINE.
  - Call logic and drawing functions on all objects.
  - Run plankon density blurring algorithm.
  - Call logic and drawing functions on all fish.
  - Time step the physics simulator.
  - The tank simulator will contain a mating list, which holds Keys to fish waiting to mate.
  - The tank simulator will contain an add fish list, which holds frozen fish/brain pairs.

– The tank simulator will contain a delete fish list, which holds Keys to fish.
– The tank simulator will contain a random fish creation counter, which is an integer.

7 Interpretation Module (Fish Body)

During an AI Module's (synonymous with "Fish Brain") update period it will generate a list of commands and send them to the Interpretation module (also known as the "Fish Body"). It is then up to the body of the fish to translate that data into specific behaviors and general movement of the fish. It does this through the generation of potential fields.

7.1 Potential Fields

Potential Fields greatly simplify many tasks that the Interpretation Module must execute. For one, it allows us to layer behavior onto the fish. Instead of forcing the brain to only send one command per update to the body, we can send it five and blend them together into very fluid action. Before we would have sent "Attack" to the body and the body would blindly swim to its target and get swallowed by a larger fish. Now, the body will receive an "Attack" command along with a "Avoid Big Fish", the fish now makes a circle around the bigger fish and attacks its prey safely.

Potential fields work by assigning force vectors to everything that you wish a moving fish to take into account. Since we want the fish to avoid terrain and other objects we create a list of unit vectors pushing away from them. Each vector is then scaled, the nearer we are to the object the higher weight it will receive. Besides avoiding terrain, the AI Module tells us we want to avoid the "Bigger" fish, so we add to the list another vector pointing away from that fish. Next, the AI informed us that we should try to attack the smaller fish, again we add another vector, only now it points towards the prey. Lastly, when the AI sent us instructions it gave us a weight for each of them, we use that weight to scale their resultant vector's.

All of the vectors in the list are now added together. The result of that summation is the direction we want the fish to go. As this process is continued it produces very fluid movement with far less computational overhead then say A*. Potential fields also allow players see fish that hesitate as they try to balance eating desires with fleeing instincts. Potential fields are a win/win proposition.

7.2 Commands

Below is table of commands understood by the Interpretation Module. In no way is it an exclusive listing, rather more a sampling of them. Unless otherwise stated a command only effects the direction that fish will flow towards. Also note that an object can be a fish, chum, marker, rock or any other physical item that is in the tank.

Follow object.	Follows an object. Will maintain a reasonable distance to the object.
Go to object, point or voxel	Goes to an object, point or voxel. Gets as close as possible to it.
Flee object, point or voxel.	Moves away from the indicated object, point or voxel.
Attack object or anything (mouth & fin).	Moves toward and attacks the specified object. (Will accept an anything tag.)
Eat object or anything (mouth & fin).	Moves toward and eats the specified object. (Will accept an anything tag.)
Move forward.	Tells the fish to move forward. May be useful when the player has possession.
Stop (fin only)	Ceases all fish movement. Will backward poison other move commands.
Stop (mouth only)	Ceases all mouth moves. Will backward poison other move commands.
Evaluate Markers	Orders a body to activate the potential fields of nearby markers.
Drop Marker	Orders a body to drop a marker at this location.
Roam	Orders a fish to move to a random location.

When executing some commands, ie mouth commands like attack, there may be a delay before that action can take place. For instance, we may require a bite animation which will take half a second to finish. Naturally, a fish will not deal its damage until that animation is complete. The brain may decide to stop the attack midstream, in which no damage is dealt. To halt a partially initiated action is to not that command on the next brain update. Naturally if body is told to attack another fish and midstream is told to attack it again it will recognize that the command is the same and will not restart the attack animation.

Within our system commands are passed from the AI Module to the Interpretation Module by the Message Passing System. The message passing system does not have a direct connection with an Interpretation Module, so each command is handed first to the Tank Simulator which gives the message directly to an Interpretation Module. This may seem a round-about technique but it allows a more flexible design and makes networking simple.

Every system on a networked game contains a fish body for purposes of client side prediction, however, there is only one brain for every fish. Therefore it is necessary for all commands to be broadcast over the network to all other computers. Since every fish body on every computer will receive the same input, each fish will behave the same on all player's screens. Since packet loss rate is never zero, a server will still send out a server update for double insurance.

The documentation of commands in the Interpretation Module ends with an outline of its implementation. Naturally this implementation is a highly compacted struct designed for local and network travel. Below is a table of that

struct's members.

Fish Body ID, Key or Handle	Reference to the fish body who is meant to receive this data.
Command ID	An integer number that is unique for each command.
Priority/Weight	A floating point value used to weight this command.
Object ID, Key or Handle	A reference to the object related to the specific command.
Timer (only used by Interp. Module).	Time, in seconds, that we've been performing this action. Not sent by brain.

7.3 Attributes, Stats and Other Maintained Data

Besides obeying commands from an AI Module, an Interpretation Module also must maintain many other data members. Arguably the most important chunk of data the Interpretation Module must retain are the Fish Attributes. Fish Attributes are detailed in the GDD and take up a full six pages. The Interpretation Module is the Gate Keeper of these data members and will guard them jealously. Any entity that wishes to change the Fish Attributes will ask the Interpretation Module for a copy of them. Once an entity makes any needed changes to them the Fish Attributes are sent to and accepted by the Interpretation Module.

In addition to the Fish Attributes, the Interpretation Module will need to maintain a robust number of statistics on a given fish. Note that statistics in the GDD were a representation of Fish Attributes to the Players, but in this instance we are referencing to numerical values that keep track of the actions of a fish. Statistics allow the Fitness Functions, AI Modules, and any other game construct to make assumptions about this fish. The body tracks these members and updates them as it detects their triggers. A list of some of these statistics are presented for further enlightenment.

- Kill rate and count
- Voxels visited
- Average speed
- Distance traveled
- Offspring produced
- Offspring rate
- Chum eaten and rate
- Plankton eaten and rate
- Total food eaten

- Total food rate
- Attacked count and rate
- Attacking count and rate
- Picked up count and rate for each object.
- Dropped count and rate for each object.
- Carrying count and rate

The actual implementation of both Fish Attributes and Statistics will, obviously, take the form of classes internal to the Interpretation Module. The classes will have the standard constructs of protected members and accessor functions. These classes will not be alone, besides Attributes and Statistics the Interpretation Module will maintain other data useful for the Graphics, Sound and Other Modules. Below is another table explaining some of the errata data archived by the body.

Graphics Client Reference	A mediator between the Graphics Class and the fish.
Sound Client Reference	A mediator between the Sound Class and the fish.
Attacked Flag	Simple flag to let the AI know we have been attacked.
Attacker's Stat Queue or Stack	Stores the stats of all the fish that have attacked this fish (learning process may clear this)
Attacker's Stat Queue or Stack	Stores the stats of all the fish that this fish has attacked (learning process may clear this)
A previous set of Statistics	Archives the statistics from the previous brain update for the AI to compare.
Fish Voxel	Calculated in reference to the fish's position.
Fish Position	Only modified by the Physics Simulator.
Current Direction(3D Vector)	Current direction this fish is trying to go.
Current Acceleration	Accel is the "Gas pedal" that the physics uses to move the fish.
Actual Speed (3D Vector)	A 3D vector that represents a fish's speed (set by physics system)
Fish's ID, Key or Handle	Needed so others know who we are in the world.
Fish Brain's ID, Key or Handle	Lets the world know who thinks for us.
Key to the Phy. Sim. object	Key to the shape that we represent in the Phy. Simulator.
Euler angles of the fish	Pitch and Yaw that this fish needs to be translated by.
Command List	List of all current commands. divided into completed and pending.

7.4 Class List

Below is a basic list of classes used by the Interpretation Module to implement its logic.

Aquarius::InterpModule	Mother Class of the Interpretation Module.
Attribute Generator	Class that creates Fish Attributes and makes sure they are legal.
Fish Attribute	Class that holds the data created by the Fish Attribute Generator. Provides Std. interface to data.
Statistic	Class which maintains the statistical data of the fish. Updated by other classes.
CmdInterp	Each command that the brain could send has a matching CmdInterp class. It performs any logic for the command.
CmdInterp Client	Provides the CmdInterp classes with a standard interface into their mother Aquarius::InterpModule.

8 Physics Simulator

The Physics Simulator is responsible for correctly simulating the interactions between spheres, cylinders, boxes, and a single height map. Objects in the simulation each have a unique identifier, known as a key, and can have such statistics as position be altered through calls that only use the object's key to access it. The physics simulator will be capable of creating and processing mementos, which would save the internal state of the simulation.

8.1 Definition of Terms

Client – An object which is using an interface of the physics simulator.
Impulse – Any physical interaction that exhibits forces so strong and sudden that it can be approximated in a single frame, such as two spheres bouncing off of each other.

8.2 Primary Interfaces

- The physics simulator requires no special initialization.
- A time step function will be provided, which will increment the simulation.
- A client may specify a rectangular boundary for all objects in the simulation, there is no boundary by default.
- A reference to a single height map object may be passed in and used in future simulation of object interactions.
- An object may be instantiated by passing in a Key that the new object will use as label for itself until its destruction.
- An object may have any of its properties be modified by a client calling an accessor function that accepts a Key and a vector or similar parameter that will be used to alter the relevant object property.
- Clients will be able to instantiate and modify:
- spheres
- cylinders
- boxes
- A method to destroy any object when given a corresponding Key.
- Multiple force vectors and torque values may be applied to the same object, along with an interface to clear acting forces.
- A gravity vector may be set.
- A water friction constant may be set.

- The physics simulator will be capable of creating and processing both network and file mementos.
- Clients will be able to cast rays in simulation and receive information about what objects they hit if any. What objects the ray would hit would also be configurable, such as only colliding with the height map.

8.3 Dependencies

- The height map class.
- The vector and matrix classes.
- The standard math library.

8.4 Simulation Methods

The first simulation method to be implemented will be "fuzzy", meaning that no impulses will be possible. The basic concept is that the forces that attempt to separate intersecting objects become exponentially stronger as the volume of the intersection increases. A way of visualizing this system would be if one were to have a pool table right before a break; one would see a set of pool balls slightly sinking into the surface of the table. As the cue ball were to hit the pool ball formation, each ball would intersect with its neighboring pool balls until they would separate by slowly pushing each other away.

The second simulation method to be implemented will be "hard", and in this case will be an impulse based system; all possible collisions will be calculated, and the earliest collision will be carried out and then another check will be made for more interactions between objects in the simulation. This method, when coded properly, would alleviate the problem of objects intersecting. This method is fairly risky to implement, as objects can fail to collide and slip through each other. An attempt will be made to properly implement this method before the pre alpha deliverable, but there is no guarantee that this technique will be completely bug free by that time. A way of visualizing this system would be if one were to have a pool table right before a break; one would see a set of pool balls sitting on top of the surface of the table. As the cue ball were to hit the pool ball formation, each ball would not intersect with its neighboring pool balls and they would all simultaneously split off in different directions like real pool balls.

A possible third technique would be to combine the two techniques mentioned above, namely such that the "fuzzy" method could slowly correct any mistakes made by the "hard" method. This technique would most certainly be slower.

8.5 Internal Details

- The time step function will use the global scope "game time step" variable in order to simulate at a constant rate. If the game time step is above

a certain threshold, say a tenth of a second, then it will be ignored and the threshold value will be used in its place so that the physics simulation may stay accurate.

- There will be a storage container for each object type, so as to speed up the processing of many objects.
- An STL map will be used to augment the Key lookup process that occurs when a client accesses an object in the simulation.
- When processing "fuzzy" physics, each frame will consist of using a single quadratic pass on the objects to figure out what forces are acting upon them that frame, and then running a single linear pass to apply the forces and velocities for the frame.
- When processing "hard" physics, each frame will consist of using a single quadratic pass on the objects that will compile a list of collisions that will occur during the frame. The earliest collision is resolved first, moving the involved objects involved to their proper locations immediately after the collision, and then recomputed against the entire world for any additional collisions; any positive results are put back onto the collision list after all objects referred to by the collision list and recomputed against the entire world. This process is repeated until the collision list is completely empty, then a single linear pass is made that moves all objects to the extent of their movement for the frame.
- Impulses will be calculated by performing intersection calculations that will generate the time, normals, and exact position at which the two objects collide at. Each possible collision between different object types will use a different function that specializes in finding the earliest collision conditions; these functions will be available as protected members of the physics module.
- Objects will be able to obey rules regarding inertia in order to allow them to rotate in response to collisions.
- There will be voxels that contain lists of all objects whose centers of mass are contained by that particular voxel, this will allow for optimizations for any objects which are smaller than the predefined voxel size.

9 Height Map

The height map object stores a collection of vertex height values that are implicitly arranged in a grid. Each square of the grid is also implicitly split into two triangles that are divided along the positive diagonal, that is to say, the vertex at [x][y] will always be connected to [x+1][y+1] by a triangle edge. Each triangle can be indexed by its "lower vertex", and an identifier for X major or Y major; as an example, the triangle with lower vertex [x][y] and an X major would be defined by the vertices [x][y], [x+1][y], and [x+1][y+1]. Unit normals will be computed and stored for each triangle.

9.1 Definition of Terms

Lower vertex	– the vertex of a triangle that has the smallest indices.
X major triangle	– a triangle whose third vertex is obtained by incrementing the x index of its lower vertex.
Y major triangle	– a triangle whose third vertex is obtained by incrementing the y index of its lower vertex.

9.2 Primary Interfaces

- Read the value of a vertex.
- Write the value of a vertex.
- Read the value of a triangle's normal.
- A function that will cause the height map to render itself by using the graphics module.

9.3 Graphical Aspect

- The aquarium's terrain will be represented by a height map, which will be detail textured to make it look more interesting.
- To distinguish between different heights, there will be multiple textures blended together at different rates, depending on the slope of the height map. To keep the height map from being too slow through texture changes, all the textures will be contained within one bigger texture.

9.4 Dependencies

- The graphics module.
- The vector and matrix classes.
- The standard math library.

9.5 Internal Details

- A single texture will be stored.
- Three subtextures will be stored.

10 Graphics

All 3D graphics and Windowing will be done by the graphics engine. The graphics engine will create a window and manage its message loop. The Window handle (HWND in Windows) will be available for anyone who needs it, and a message will be sent out every time any time the HWND changes, which should only happen at most when the user changes between full screen and windowed mode.

The 3D Graphics library will make heavy use of the linear algebra library for viewport transformations. The library is coded for OpenGL, which makes it inherit some of OGL's restrictions such as:

- Textures' dimensions must be powers of two. It is okay for the textures to not be square.
- More then 8 lights are not allowed in the scene.

Not all of OpenGL's designs are copied, for example the coordinate system is right handed as this is more natural to most people. It will be pointed out when there is a significant difference between the way OpenGL and the Graphics engine do the same thing.

10.1 Graphics Clients

Most other modules will communicate to the Graphics module through a Graphics Client. A graphics client contains all the information needed to render a specific type of object such as a fish or a button. The client will be able to animate the model between different frames, so if a piranha was biting a shark causing the shark to die, the shark's graphics client will cause it to turn belly up.

A graphics client will request from the Driver the current viewable area of the world so that it can cull itself in some fashion. All graphics clients should be derived from the graphics client base class described below.

10.1.1 Interface

- Render (private)
  - A virtual function to override. Render will be called by the Driver when it has determined that it is the "best" time to render the object.
- SetRender
  - A non-virtual function to override. SetRender should be called by the holder of the client if they want the object to be drawn. SetRender will decide if it should be rendered or not.

10.1.2 Implementation Details

Each graphics client needs to be able to know the current state of the object it is to draw. Additional functions such as "SetFrame", "Animate", and "SetOrientation" are suggested to be added, even though they are not required.

10.1.3 Implementation Concerns

- While virtual functions are usually touted as having almost no overhead, will this small amount be negligible?

10.2 Driver

The other main class that objects will communicate to is the Graphics Driver. Most interaction to the Driver will be to request other objects (i.e. Models, Textures) and to set basic rendering values (changing the fog color, the background color). While you can render both colored triangles and colored quads through the driver, it is not recommended that you do so, as this bypasses the interfaces set up, and takes away the possible speed advantages.

10.2.1 Interface

- Initialize
  - Sets up the graphics engine so that it can create a window. Will enumerate display modes.
- Destroy
  - Cleans up everything the graphics module allocated
- CreateWindow(width,height,bpp,fullscreenflag)
  - If fullscreenflag is set, changes the resolution to width x height x bpp.
  - If fullscreenflag is not set, creates a window with client area width x height.
  - Does not change the resolution.
- Flush
  - Renders all drawing set to be rendered.
  - Also does any background processing the OS requires on the window.
- SetZBuffer(flag)
  - If flag is true, enables using the Z-buffer for hidden surface removal.
  - If flag is false, disables using the Z-buffer, so things get drawn using painter's algorithm.

- SetFog(flag, r,g,b)
  - If flag is true, enables linear fogging to the color r,g,b.
  - If flag is false, no fogging is created.
- SetBGColor(r,g,b)
  - Sets the background color to r,g,b.
- SetView(position, lookat, up)
  - Sets the viewing matrix to have the view plane centered at position, be looking at the point lookat, with an assumed up vector of up.
- DrawTriangle(vertex1,vertex2,vertex3, r,g,b)
- DrawQuad(vertex1,vertex2,vertex3,vertex4 r,g,b)
- DrawQuad2d(vertex1,vertex2,vertex3,vertex4,r,g,b)
  - Draws a triangle, quad, or 2d quad of the specified color.
- SaveScreenshot(filename)
  - Saves a screenshot of the current frame of animation to the file specified.

10.2.2 Implementation Details

- Stores an array of linked lists (or an equivalent accessible interface) to allow objects to be drawn sorted by texture type and drawing order.

10.2.3 Implementation Concerns

- Rendering order is not well defined between sprites and primitives explicitly drawn. Currently, it is expected that the drawing order will be as follows:
  1. Draw the Height Map
  2. Draw the 3D objects
  3. Draw the HUD (with Z-buffer disabled)
  4. Draw the primitives (with Z-buffer disabled)

  This may cause rendering irregularities, but changing this may cause the primitives to z-fight with the HUD.
- Not separating the message pump from the flushing could cause Windows to assume that the program is not responding.

10.3 Model

All static 3D models will be cached in an optimized form for drawing it. The model will need to hold the vertices for each triangle, along with the texture coordinates for each triangle.

10.3.1 Interface

- Draw(position,orientation,texture1)
- Draw(position,orientation,texture1,texture2)
  - Draws the model immediately at the specified orientation, position, with the specified textures.

10.3.2 Implementation Details

A Display List should be used to cache the model data.

10.3.3 Implementation Concerns

- Is there any easy way to add hierarchial models?

10.4 AnimatedModel

A problem with the Model class is that only static models can be stored. If any animation is desired, the AnimatedModel class should be used. It extends the functionality of the Model class with the following interface:

10.4.1 Interface

- Animate(timestep)
  - Steps timestep units through the animation.
- SetNextFrame(frame)
  - Sets frame as the next frame to interpolate to.

10.4.2 Implementation Details

Vertex linear interpolation is currently the expected method to change between frames. Later, different interpolation techniques may be added.

10.4.3 Implementation Concerns

- Vertex linear interpolation may look really bad if not enough keyframes exist.

10.5 Texture

One of the best ways to add detail to a 3d object is with textures, which is supported in hardware by all modern video cards. While the Texture class does not do much on its own, it serves as a storage device for the textures which can be passed around. When a model specifies that it wants to be drawn with TextureA and TextureB, it will pass Texture objects (not to be confused with OGL texture objects), that hold the data needed to grab the correct texture.

10.5.1 Interface

- Constructor(filename, mipmapflag)
  - Loads the file from filename into the texture.
  - If mipmapflag is set to true, the the texture will be mipmapped, otherwise the texture will not be mipmapped.
- LoadFrameBuffer
  - Loads the data from the frame buffer into the texture. This is mainly expected to be used for creating screenshots.

10.5.2 Implementation Details

OpenGL requires a texture to be a power of 2 in both width and height (64, 128, 256), but the texture does not have to be square. If the file specified does not satisfy these requirements, then the final strip will be stretched out to allow filtering and mipmapping to work best.

10.6 SubTexture

A slow operation in OpenGL is changing textures. This suggests a simple optimization – pack textures which are commonly used together into one bigger texture. This way changing the current texture does not need to be done as often. The SubTexture is an encapsulation of this optimization. It extends the Texture class with the following functionality.

10.6.1 Interface

- Constructor(Texture, sLow, sHigh, tLow, tHigh)
  - Creates a SubTexture out of texture with s coordinates ranging from sLow to sHigh and t coordinates ranging from tLow to tHigh.

10.7 Font

When printing out characters, it is much easier to access printf style functionality instead of explicitly drawing quads with specific texture coordinates. The Font class simplifies the writing of text to the screen by loading a font file and providing a simple printf style function to display characters with.

10.7.1 Interface

- Constructor(file)
  - Creates a font from the specified file
- Write(xPosition, yPosition, string, data, ...)
  - Writes text at normalized screen coordinate (xPosition, yPosition) as if printf was called:
    printf(string , data , ...)

10.8 WideFont

WideFont is an extension of font that prints out wide character strings, to easily allow multi-lingual support. The Write function is different in that it takes a wide character string (wchar_t[]) instead of a a normal string. Other than this difference it is functionally the same.

10.9 Sprite

When drawing the HUD, it is unnecessary to specify a Z coordinate, as everything is supposed to be floating in a 2D plane. A sprite has a texture which is saved.

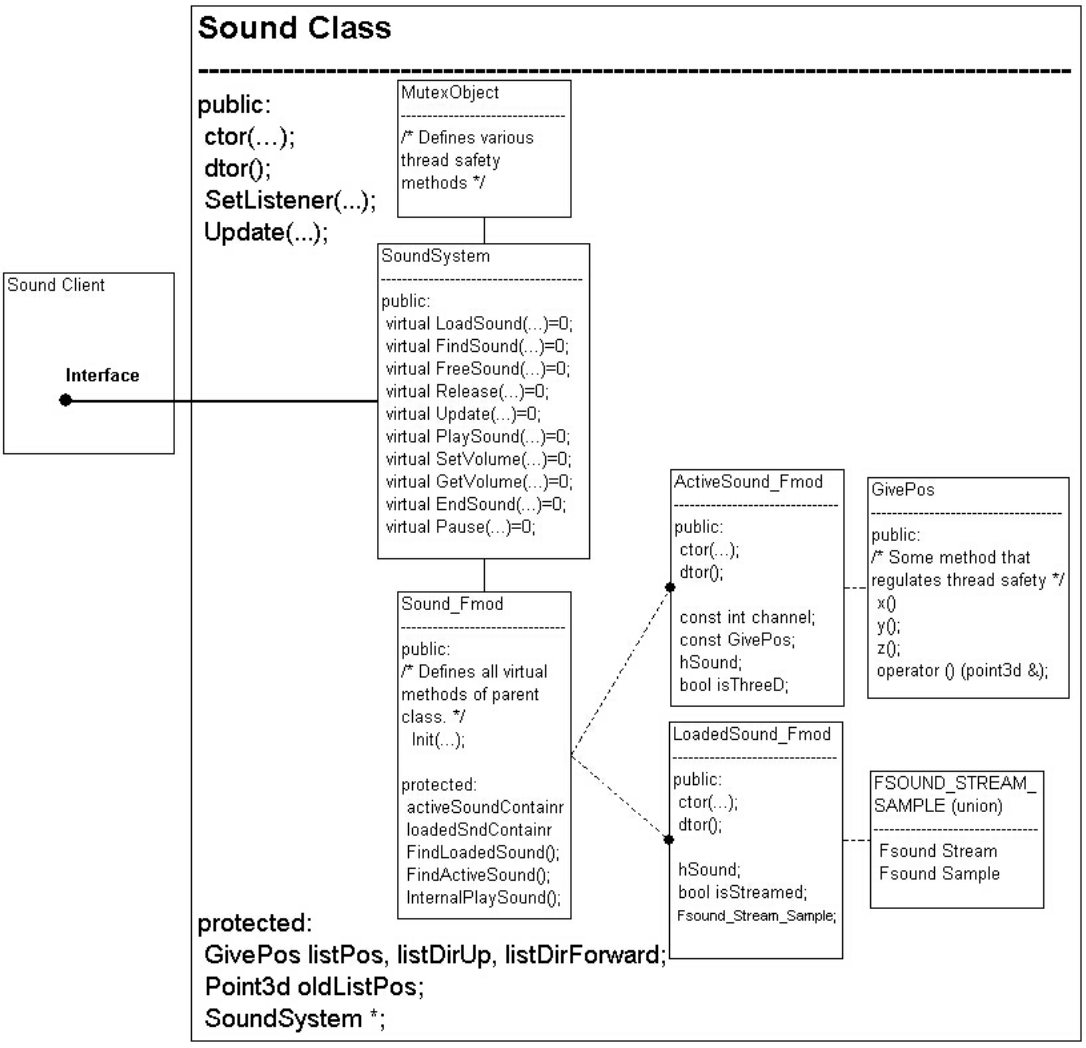
10.9.1 Interface

- Constructor(Texture)
  - Set the texture for the sprite.
- SetRender(xPos,yPos,width,height rotAngle, r,g,b)
  - Renders the sprite at the specified normalized screen cordinate position.
  - The rendering engine will choose an optimal time for rendering the sprite.

10.10 Sprite3D

With particle effects becoming an extremely popular effect to add, a 3D sprite, often called a "point sprite." will be supported. It will act the same as the the Sprite class, except that SetRender will take an additional zPos, which is how far into the screen the Sprite3D will appear.

11 Sound



As shown in the above diagram the Sound class serves as a generalized wrapper into the FMOD sound library. A casual glance at the hierarchy begs the question, "Why have the extra two levels of abstraction? Why not scrap the Sound and SoundSystem classes and allow client to direct access to Sound_Fmod class?" The battle cry of this game design is abstraction, modularization and extendibility, this concept is fulfilled in the Sound Module design. By adding two extra layers of abstraction it makes it trivial to add another sound library choice for end users of the game.

The question then becomes, "Why two layers and not one?" Simply put two layers were chosen because of future considerations. In the event that some sound libraries may require special consideration in certain areas (ie loading, initialization, etc.) the sound class is meant to act as the regulator between the user and the SoundSystem class. Thus, if later on, another sound library is added to the design of the system it should require almost no change to the client code of the game.

11.1 Sound class

As mentioned previously, the Sound class is the outermost wrapper into the inter-workings of whatever sound library the game chooses to use. It also handles several key points for the SoundSystem class, initialization and listener/update control. This frees a client of the Sound Module from worrying about specific library initialization code and from filling out cumbersome parameters during an update. Below is a detailed description of all interesting public methods. All omitted methods are total wrappers around a SoundSystem method and are completely documented in Sound.h.
Sound(SoundOutput type = FMOD);

- Default ctor for the Sound class. It forces the creator to choose the type of sound library the client will use for the life of this object.
- Parameter is an enum instructing the Sound Class what type of sound library to work with. Currently only support FMOD. Overloaded to assume FMOD.
- Will throw an exception if the type parameter is set to any value but FMOD.

```
void Sound::SetListener (GivePos * pos, GivePos * dirUp,
                         GivePos * dirForward) throw ();
```

- Must be called at least once and provide the Sound class with the position and direction of the listener. Must be called again if the listener ever changes. 3D Sound will only work if BOTH are set.
- A GivePos is a class that allows the sound class to directly query an object. Is detailed later.
- dirUp must point to an object that returns the upward facing vector of an object, while dirForward returns forward facing vector.
- Caller may pass NULL to any value but they must fill them all before sound output will work.

void Update(**float** elapsedTime) **throw**();

- Must be called once a frame so all 3D and streamed sounds can be updated.
- Parameter is the amount of time since the last time update was called (in seconds). Must be true time, not game time or offline time.

```
bool Sound::LoadSound (const char * pathName, HSound & h,
                       int loadOps) throw ();
```

- Preloads a sound for the caller. If the sound is already loaded this call will NOT fail. It reserves the right to either load another copy of the sound or give the caller a reference to the already loaded copy.
- Second parameter is the var that will be filled with a handle to the sound. Detailed later.
- Returns false if something goes wrong. Will NOT return false if the sound is already loaded. Common failure would be that the max number of sounds are already loaded. Another common fail is if the pathName is longer then the def'd value of MAX_SOUND_FILE_PATH_NAME-1. Returns true if all goes well.
  - Load Options

SOUND_2D	Force 2D treatment of this sound.
SOUND_3D	Force 3D effects of this sound.
SOUND_LARGE	Suggest streaming to the internal class.
SOUND_SMALL	Force non-streaming to the internal class.
SOUND_LOOP_OFF	Leave looping off for this sound.
SOUND_LOOP_ON	Turn looping on for this sound.

11.2 HSound

Whenever a client loads a sound they are required to pass a reference to an HSound. That HSound is filled with a handle to that particular sound. From that point on if the client needs something done to that loaded sound, be it playing or pausing, they must then pass that same HSound as a parameter. The HSound must be retained by the caller since the Sound Module reserves the right to add reference counting to the sound system in the future.

11.3 SoundSystem Class

The SoundSystem class lays down an outline of what all sound libraries must try to conform to. Specific documentation on method behavior must be provided by the derived class. Notice however that no set init function is required by the soundSystem. Each class may design one that suites their sound library best. Below follows a listing of all the methods in the SoundSystem class, all of which are pure virtual and must be defined by children classes.

```
virtual bool LoadSound(const char *pathName, HSound &h,
                       int loadOps) = 0;
virtual bool FindSound(const char *pathName,
                       HSound &h) = 0;
virtual bool FreeSound(const HSound &h) = 0;
virtual bool Release(void) = 0;
virtual bool Update(UpdateData &up) = 0;
virtual bool PlaySound(HSound &h,
                       bool StartPaused = false) = 0;
virtual bool PlaySound(HSound &h, const GivePos &pos,
                       bool StartPaused = false) = 0;
virtual bool SetVolume(HSound &h, int volume) = 0;
virtual bool SetVolume(int volume) = 0;
virtual bool EndSound(HSound &h) = 0;
virtual int GetVolume(HSound &h) = 0;
virtual int GetVolume(void) = 0;
virtual bool Pause(HSound &h, bool SetPaused) = 0;
```

11.4 Sound_Fmod Class

Currently, our system only supports FMOD sound output. The class that links the Sound class with FMOD is Sound_Fmod. It derives publicly from the SoundSystem class and fills out all of its virtual methods. Below follows a listing of all the public members in Sound_Fmod class.
bool Init(**const** FmodOps *ops=NULL);

- Init method for Sound_Fmod. Must be called for sounds to work.
- FmodOps is a struct will all fmod related initialization information. The caller may pass NULL for this and the function will use default values.
- Will return false if fmod init functions fail or if channels are set to zero.
- Reserves the right to throw exceptions.

bool Release (**void**) **throw** ();

- Closes down the entire sound system and frees all the memory used by the class.

bool LoadSound (**const char** *pathName, HSound &h, **int** loadOps) **throw** ();

- Preloads a sound for the caller VIA FMOD. If the sound is already loaded this call will NOT fail. It will instead load another copy of the sound.
- PathName is the file name for the sound to be loaded. can NOT be larger then MAX_SOUND_FILE_PATH_NAME - 1 else the method will return false.

- second parameter will be loaded with a handle to the sound.
- See Sound::LoadSound method in sound.h for details on how to fill this param out.

bool FindSound (**const char** *pathName, HSound &h) **throw** ();

- Attempts to find a copy of an already loaded sound.
- HSound will be loaded with a handle to the sound.
- Returns false if it couldn't find the sound. Returns true if it did find the sound.

bool FreeSound (**const** HSound &h) **throw** ();

- Frees a preloaded sound.
- Returns false if it couldn't find the sound.

bool Update (UpdateData &up) **throw** ();

- Must be called once a frame so that the positions of all the 3D sounds in the system.
- UpdateData is a struct with all data fmod needs to update the system.

```
bool PlaySound (HSound &h,
                bool StartPaused=false) throw ();
```

- Plays a given sound, forever if the sound was loaded to loop or just once otherwise..
- Users can set last parameter to true if they want the sound to start up paused. Defaulted to false.
- Returns false if the handle is invalid or something else goes wrong. Common failure would be that the max number of sounds are already playing.

```
bool PlaySound (HSound &h, const GivePos &pos,
                bool StartPaused=false) throw ();
```

- Plays a given 3D sound. 3D effects will not work if the sound wasn't loaded with the 3D options set. Beyond 3D effects, behaves just as Sound_Fmod::Play(HSound &, bool);

bool SetVolume (HSound &h, **int** volume) **throw** ();

- Alters the volume of the given sound relative to the global sound. It will alter the volume of ALL active sounds with the given handle.

- Last param is a 0(silent) to 255(full on) value for the new volume.

int GetVolume (HSound &h) **throw** ();

- Returns the volume of the given sound. If there are multiple instances of this sound will return the value of the first that is found.
- Returns the current volume of the given sound. Range will be 0(silent) to 255(full on). If something goes wrong will follow fmod's example and returns 0 to the caller.

bool SetVolume (**int** volume) **throw** ();

- Alters the global volume.
- Volume values must be between 0(silent) and 255(full on).

int GetVolume (**void**) **throw** ();

- Returns the global volume.
- Returns the current global volume. Range will be 0(silent) to 255(full on). If something goes wrong will follow fmod's example and returns 0 to the caller.

bool EndSound (HSound &h) **throw** ();

- Stops all instances of a sound from being played.

bool Pause (HSound &h, **bool** SetPaused) **throw** ();

- Sets the paused flag to either true or false.
- Returns false if the handle is invalid or something else goes wrong.

11.5 Sound Clients

The technique for interacting with the Sound Module is through a client. A client can be individually tailored to control the sound logic for a specific object. The Sound class itself does not implicitly provide these clients since each is very specific for each object, yet they will be written by the same programmers. The following table shows where our game will use Sound Clients and what each client is responsible for.

Interpretation Module Client	Landing Attack sounds, Eating sounds
Game Object Client	3D Sound nodes (bubbly and wave noises).
UI Module Client	Affirmative click and Bad buzzer. Background music.
Debug Client (optional)	In the future may add debugging noises for developers.

When a sound client polls its dependant it creates a GivePos class. A GivePos class allows the Sound class to have random access to any positional data of an active sound. GivePos classes are completely localized to the Sound Clients and no other module will need give them any consideration.

Lastly, the following file types are supported by the Sound Module; .WAV(PCM and Compressed), .MP2, .MP3, .OGG, .RAW, MIDI, .WMA(streaming only) and .ASF(streaming only). In the future, any new sound library added to the system must guarantee the support of .WAV(PCM) and .MP3.

12 Menu System

The menu system will consist of three primary classes that all other GUI elements can be derived from. These are the Screen class, the MenuItem class and the Menu class. The GUI Handler object will handle these.

12.1 The UI Handler

The UI Handler will be the primary interface for all the GUI screens in the game. It will create and own the different screens and switch between them when directed to do so.

12.2 Public Interface

void Initialize ();

- Creates all GUI screens

void Logic();

- The UI will handle switching between screens based on input from the player.

12.3 Screen Class

The Screen class is the class that defines the GUI of the current in-game screen. It will contain all the menus for that screen, handle input from the player input module, handle the focus and handle the mouse cursor.

Input Handling

When the screen receives input messages, it will first check if there is a menu item with the focus. If so that menu item will receive the message. If not then the screen will send mouse click messages to the menus and keyboard messages to the key map. If there is a menu item with focus and that menu item relinquishes the focus upon receipt of the message then the screen will send the message to the menus as usual.

Focus

The screen will have a pointer to the menu item with the current focus. Since each menu item has a handle to their screen they will be able to set themselves as the current focus. Once a menu item has the focus it will receive all input messages from the screen until it relinquishes focus. Once it has relinquished focus the current input message is sent to the menus as usual.

Mouse Cursor

The screen will have a mouse cursor object. Mouse related messages will be relayed to the mouse cursor. The position of the mouse cursor and not the actual mouse will be relayed to the menus.

Key Map

The screen will have a key map object. This will be an invisible menu that handles mouse messages and keyboard input when there is no other menu item in focus. It will consist of invisible menu items that send messages to the message handler when invoked.

Tool Tip

The screen will have a pointer to a tool tip. This will be cleared at the start of the Screens process cycle. Menu items will be able to set the tool tip when they get a mouse over message. The tool tip is drawn before the Mouse Cursor but after all other GUI items.

Pull Down Menu

The screen will have a pointer to a Pull Down Menu. This is a special menu that needs to be drawn before the tool tip but after all other GUI objects.

Communication to game

Each screen will read the status of its buttons and send relevant messages to the game via the message handler.

12.3.1 Public Interface

void takeInput(InputMessage message);

- Used by the PlayerInput module to send input messages to the UI.

void SetFocus(MenuItem∗ item);

- Used to set a menu items focus.

void ClearFocus();

- Used to clear the Focus.

void SetToolTip(ToolTip∗ tip);

- Used to set the current Tool Tip.

void SetKeyMap(KeyMap∗ map);

- Used to set the current KeyMap object.

void SetPullDown(PullDownMenu* menu);

- Used to set the current pull down menu.

bool addMenu(Menu* menu);

- Used to add a menu to the Screen.

void draw()

- Causes screen to draw all menus

12.3.2 Public Class Definition

```
class Screen
{
public:
    void takeInput(InputMessage message);
    void SetFocus(MenuItem* item);
    void ClearFocus();
    void SetToolTip(ToolTip* tip);
    void SetKeyMap(KeyMap* map);
    void SetPullDown(PullDownMenu* menu);
    void draw();
};
```

12.4 MenuItem Class

The MenuItem class will be the base GUI unit. All GUI elements will be derived from this class.

Handle to Screen

Menu items will have a handle to the screen. Through this they may communicate messages to their screen.

Handle to Menu

Menu items will have a handle to the menu they are on. This will allow them to communicate up one level.

Input Handling

Menu Items will be able to receive GUI messages. They will handle messages in their own way. If a menu item processes a message, it sets its update flag.

Tool Tip

Each menu will have two tool tips that they may use to display information when they receive a mouse over message. These are the short tool tip and the long tool tip. The short tool tip gives only basic information, usually just identification, and is displayed when a mouse-over message is received. The long tool tip gives more detailed usage information and is displayed after a period of time has elapsed. When a mouse-over message is received the menu item can set the current tool tip through its screen handle.

Dimensions, Captions and Bitmaps

Menu items will have height and width dimensions as well as a position point that refers to the upper left hand corner of the menu item. They will also have an optional bitmap and a caption that they may use for display purposes.

Drawing

The menu item will be responsible for drawing itself when its draw method is invoked.

12.4.1 Public Interface

virtual void takeInput(InputMessage message);

- Takes UI message.

void setPositiont(x,y);

- Sets the position.

virtual void draw();

- Tells MenuItem to draw itself

12.4.2 Public Class Definition

```
class MenuItem
{
public:
    virtual void takeInput(InputMessage message);
    virtual void draw();
    void setPositiont(x,y);
};
```

12.5 Menus

Menus are a derived class from menu-items. The distinguishing features of menus are that they have a list of menu-items. And that they pass messages that they receive and do not handle on to their menu-items.

Input Handling

When a menu receives messages, it first checks to see if it can process the message, if not the message is forwarded to its menu items. If the message is a mouse click, the click is checked against each menu item until a collision is found and then the click is forwarded to that menu-item.

Control of Menu Items

Menus will be able to set the positions of their menu items. This will allow menus to scroll their menu items or drag them if the menu is dragged.

Drawing

When the menus draw method is invoked, it draws itself and then it calls the draw method of all its menu items.

12.5.1 Public Interface

virtual void takeInput(InputMessage message);

- Takes UI message.

void setPositiont(x,y);

- Sets the position.

virtual void draw();

- Tells MenuItem to draw itself

12.5.2 Public Class Definition

```
class Menu : public MenuItem
{
public:
    virtual void takeInput(InputMessage message);
    virtual void draw();
    void setPositiont(x,y);
};
```

12.6 Mouse Cursor

The mouse cursor is a special GUI object that displays the mouse. The mouse cursor will handle screen messages and be able to draw itself.

12.6.1 Public Interface

void setPosition(x,y);

- Sets the position of the cursor.

virtual void draw()

- Draws the mouse cursor.

12.6.2 Public Class Definition

```
class MouseCursor
{
public:
    void setPosition(x,y);
    virtual void draw();
};
```

12.7 Key Map

The key map is a special UI object that maps game actions to keys. It is owned by a screen that will send messages to it from the player input module when no object is in focus. The key map will then translate these messages into game messages.

12.7.1 Public Interface

virtual void takeInput(InputMessage message);

- Takes an input message
- Uses Message Handler to send game message

void mapKey(**int** keyID, **int** actionID);

- Takes a player input module key constant
- Takes a UI action ID.
- Allows a key to be mapped to an action.
- Defaults will be pre-mapped.

int getKeyID(**int** actionID)

- Gets the player input module key ID of the action specified by actionID

12.8 Derived GUI Objects

Text Box: Menu Item Text boxes are a menu item that display text.

Tool Tip: Text Box Tool Tips are a special designation of text boxes.

Edit Box: Text Box Edit boxes are text boxes that process keyboard input. They do this by processing alphanumeric keyboard messages. These messages are translated into characters that are stored in a string. The string is echoed back to the screen. The input string is made available to a requestor.

Push Buttons: Menu Item Push buttons are menu items that process mouse clicks.

Bar Graphs: Menu Item Bar graphs are menu items that read the currently selected fish and display relevant statistics in bar graph form.

Check Boxes: Menu Item Check boxes are menu items that toggle either checked or unchecked with mouse click messages.

Hierarchical Check Boxes: Menu A hierarchical check box is a check box that is a menu of check boxes. If it is checked then all check boxes on the menu are also checked.

Pull-Down Menu Bar: Menu Pull-down menu bars are menus that only display their caption or bitmap until they are selected. When they are selected, they register themselves with the screen so that they can display their full menu.

Slider bars: Menu Slider bars are menus that have a dummy menu item that it uses to mark the current slider value. This value will be directly readable or it can be sent to the game via the screens message handler reference.

Slider-Value Combo bar: Menu A slider-value combo bar is a menu that contains a slider bar and an edit box. When the slider is adjusted, the slider-value combo adjusts the text in the edit box. When the edit box is changed, the slider-value combo bar adjusts the position of the dummy menu item on the slider.

12.9 Console

The console is a special GUI object. It will be a screen with a menu that will have a text box and an edit box. The text box will be used to display system messages from the debugger module. The edit box will be used to send messages to systems within the game. For example, the debugger has several options that can be set using a string of commands. The commands that can be sent are up to the system. The console object will simply read the string and dispatch it to the appropriate system based on the dispatch command received. For example, the string "\Debug +SND GFX" will send the string "+SND GFX" to the debug module which will interpret that string as it sees fit.

Console commands are:

\Debug	Dispatch remaining string to debugger
\AI	Dispatch remaining string to AI system
\NET	Dispatch remaining string to Networking
\GFX	Dispatch remaining string to graphics module
\PHY	Dispatch remaining string to physics module
\SND	Dispatch remaining string to sound module
\MSG	Dispatch remaining string to message handler
\GUI	Dispatch remaining string to screen handler
‘	Toggle console display Not remappable

13 Threading

Threads are unavoidable for networked games, yet as the thread count increases the risk of overly complex class hierarchy also rises. Complexity is definitely something to avoid, so great care is taken to select logical places to introduce threading. The three threads we have selected are isolated from the rest of the game to avoid complications. This isolation is done by structuring threads to inject data into the main thread rather then blindly changing variables in the global program scope. Of course actual thread design is specific to each module, however, each will use either the message passing system, module specific queues, or general thread safety mechanisms (see mutexes) to insure isolated threads.

13.1 Threading Locations

Excluding the main thread, the most obvious place to introduce a thread is of course the networking. The AI will use the other thread for Learning/Training. Training is an excruciatingly slow process which would certainly lag our game if left on the main thread. The following table restates all game threads and details their safety technique(s).

Networking thread	Socket Listening, etc.
	Module specific queue, mutexing.
AI thread	Training/Learning.
	Mutexing.
Main thread	All other modules.
	Mostly message passing, some mutexing.

13.2 Thread Implementation

For the actual implementation of threads we will use the Boost C++ extension library. Boost is a freely available library for both private and commercial use that is open source but NOT under the GPL. Besides being completely free, Boost, and specifically the threading portion, is cross platform compliant and simple to use. For specific information on Boost please go to www.boost.org.

Boost provides an effective wrapper for creating a thread. All a user need do to spawn a thread is to create a Boost::thread object and pass it the address of the function where the thread will start. The only down fall to Boost threads are that they do not allow users to pass parameters to the starting function. This is unacceptable for our purposes so we will create a ThreadGroup object as a wrapper around Boost threads. It will provide threads with all the flexibility of Boost while adding an initialization parameter should they need it. Below is a list of all public methods of the ThreadGroup class .
bool AddThread(**void**((∗threadFunc)(**void**)), **void** ∗threadParam) **throw**();

- Adds a thread of execution to the system. Will only activate that thread if the last parameter is set to true.

- First parameter is a pointer to a function that takes nothing and returns nothing. This will be the point of initial execution.
- Second parameter is a void pointer to the data the user wants the thread to receive.

bool RemoveThread(**void**((∗threadFunc)(**void**))) ;

- Effectively pulls the plug on a thread.
- Does not guarantee no throw. If an exception is thrown program should immediately terminate.
- Will return false if thread is unknown.
- First parameter is a pointer to the point of initial execution aka the starting function.

void Sleep(**void**((∗threadFunc)(**void**)), **const** boost::xtime& xt) **throw**();

- Sleeps a thread for a set amount of time.
- First parameter is a pointer to the point of initial execution aka the starting function.
- For information on second parameter see documentation on boost::xtime class at http://www.boost.org/libs/thread/doc/xtime.html

void Yield(**void**((∗threadFunc)(**void**))) **throw**();

- A passive sleep call. Sleeps the thread only if a sibling thread needs a time slice.
- First parameter is a pointer to the point of initial execution aka the starting function.

void ∗GiveParam(ThreadFunc threadFunc) **throw**();

- Returns to the caller the relevant parameter for this thread.
- First parameter is a pointer to the point of initial execution aka the starting function.

For further help, here is a example of how to use the ThreadGroup class.

```
ThreadGroup g_TG;

void myThreadFunc(void)
{
    MyDataType *data = g_TG.GiveParam(myThreadFunc);

    while(1)
```

```
    {
        //do stuff....
        //...

        //take a break
        g_TG.Yeild(myThreadFunc);
    }

    return;
}


int main(void)
{
    //insatiate the data and create the thread
    MyDataType *data = new MyDataType;
    g_TG.AddThread(myThreadFunc, data, false);

    //start the thread
    g_TG.Join(myThreadFunc);

    while(1)
    {
        //do some stuff
        //...

    }

    //we're done, time to clean up the game.
    if(!g_TG.RemoveAllThreads())
        /*really bad*/;

    return 0;
}
```

13.3 Thread Safety (Mutexing)

When using threads it is important to keep them synchronized to each other. The simplest way to do this is to use mutexing. Windows mutexes can be complicated to deal with, thankfully Boost's thread library provides an exceptional wrapper around them. Just like Boost threads these mutexes are just as simple to use and are completely flexible. In fact they are so flexible that we will not create a proprietary wrapper but rather directly use the Boost mutexing system.

Within the Boost library there are six types of mutexes. For our game we will only concern ourselves with half of these mutexes. Specifically we will use

the recursive mutexes. Recursive mutexes are slightly slower but ensure tighter control of our threads. Within this set of mutexes there are three different flavors; plain mutex, try mutex and timed mutex. A plain mutex has no special facilities, when you try to lock it the thread will block until you are able to lock it. A try mutex will attempt to lock the thread, but if it is already in use it will not block. The try mutex will instead fail to lock. Last but not least, a timed mutex will try to lock for a user specified period of time before giving up and failing to lock.

Locking a boost mutex is a very simple task. To lock a thread all one need do is to create a locking object. The locking objects also comes in three flavors, normal, try and timed. If you are trying to lock a timed mutex you may use any of these objects. A try mutex cannot be locked with a timed lock, but it still can use both normal and try locks. While the plain mutex can only use the blocking normal lock. Unless a user created a normal locking object, a locking object must be checked to see if it obtained the lock. As an added bonus with Boost locking object a user is forced to implicitly unlock the object, *a locking object's dtor automatically unlocks a mutex.*

It is recommended to thread designers that they use recursive timed threads for their implementations. Recursive timed mutexes allow the greatest flexibility of locking without sacrificing size.

In the event that readers are confused at this point an example of thread safety is provided for edification.

```
#include <iostream>
#include <boost/thread/xtime.hpp>
using namespace std;

//include all the needed boost thread headers.
#include <boost/thread/mutex.hpp>
#include <boost/thread/thread.hpp>
#include <boost/thread/recursive_mutex.hpp>
using namespace boost;

//create an instance of all boost mutexes
recursive_mutex           recurMutex;
recursive_try_mutex       recurTryMutex;
recursive_timed_mutex     recurTimeMutex;

void BlockingTest(void)
{
    //this call will block until somebody else lets go of
    //the mutex.
    //notice that the timed mutex is used to create this
    //blockingLock.
    recursive_timed_mutex::scoped_lock
        blockingLock(recurTimeMutex);

    cout << "  BlockingTest thread finally locked the thread."
         << endl;
    cout << "  BlockingTest unlocks the thread in blockingLock"
         << " dtor." << endl;
}

void TryTest(void)
{
    recursive_timed_mutex::scoped_try_lock
        tryLock(recurTimeMutex);

    // check to see if we got the lock
    if(tryLock.locked())
        cout << "  TryTest SUCCEEDED in locking the mutex"
             << endl;
    else
        cout << "  TryTest FAILED in locking the mutex,"
             << " somebody already has it." << endl;
}
```

```
void TimedTest(void)
{
    //see www.boost.org/libs/thread/doc/xtime.html for doc on xtime
    xtime xt;
    xtime_get(&xt, boost::TIME_UTC);
    xt.sec += 2;                    //set wait for 2 seconds

    recursive_timed_mutex::scoped_timed_lock
        timedLock(recurTimeMutex, xt);

    if(timedLock.locked())
        cout << "  TimedTest SUCCEEDED in locking the mutex"
             << endl;
    else
        cout << "  TimedTest FAILED in locking the mutex, "
             << "somebody already has it." << endl;
}

int main(void)
{
    cout << "Main thread locks the mutex." << endl;
    recursive_timed_mutex::scoped_lock
        mainLock(recurTimeMutex);

    //check to see if we got the lock
    if(mainLock.locked())
        cout << "  Main SUCCEEDED in locking the mutex"
             << endl;
    else
        cout << "  Main FAILED in locking the mutex"
             << endl;


    cout << "Create the BlockingThread and TimedTest "
         << "threads."
         << endl;
    thread BlockingTestThread(&BlockingTest);
    thread TimedTestThread(&TimedTest);

    cout << "Create the TryTest thread, it'll also try to "
         << "lock the mutex."
         << endl;
    thread TryTestThread(&TryTest);

    cout << "Main unlocks the mutex and sleeps for 1 sec."
         << endl;
```

```
    mainLock.unlock();
    //call sleep...

    return 0;
}
```

Program Output:

```
Main thread locks the mutex.
  Main SUCCEEDED in locking the mutex
Create the BlockingThread and TimedTest threads.
Create the TryTest thread, it'll also try to lock the mutex.
  TryTest FAILED in locking the mutex, somebody already has it.
Main unlocks the mutex and sleeps for 1 sec.
  BlockingTest thread finally locked the thread.
  BlockingTest unlocks the thread in blockingLock dtor.
  TimedTest SUCCEEDED in locking the mutex.
```

14 Debug Module

The debug module will be a singleton that encapsulates a number of functions for tracking messages embedded in the code. The debugger will buffer messages and output them either at the end of each loop or when it reaches its buffer capacity. This functionality is as follows:

- Logging
  - Uses a file to log messages
- Console Output
  - Uses displays messages to an in-game console used to display debug messages.
- Standard output
  - Sends messages to standard output

The debugger will implement different levels of information. A higher debug level means that the message is more significant. Some guidelines to follow are given below.

- Level 1: Any message.
  - Often these messages are just to show that the program is progressing as desired.
- Level 2: Uncommon messages.

+	toggles on
-	toggles off
L1	level 1 messages
L2	level 2 messages
L3	level 3 messages
L4	level 4 messages
LALL	All levels
AI	AI system messages
NET	Networking messages
GFX	graphics module messages
PHY	physics module messages
SND	sound module messages
MSG	message handler debug messages
ALL	all systems
DISPLAY	Toggle messages while preserving configuration
\LOG	direct commands to logger
STANDARD	(Console Messages only) directs to standard output
SCREEN	(Console Messages only) directs to in-game console

Figure 1: Debugger Command Summary

  - Messages that the programmer wants to look out for but may not happen every loop.

- Level 3: Rare messages.
  - Messages that the programmer does not expect to see or that the programmer wants to have special significance.

- Level 4: Unique messages
  - Level 4 should be reserved for very special messages.

In addition to specifying a level, the programmer can specify what modules they receive messages from. ID's will be available for each module. The level and module ID identifiers will allow the programmer to target what messages they are receiving. Messages will also be able to be turned off or on while preserving the output configuration. Commands for the debugger are by default for the non-log messages. The "\LOG" command will direct all following commands to the debug logger.

Debugger commands will be input via a string that the debugger will parse out. The string will be set via the in-game console or an input file. Note that the input file will be read once at the start of the program. Figure 1 is a list of debugger commands.

An example of how this would work is as follows:

"+L1 +L4 +AI +SND -MSG"

This would turn on level 1 and level 4 messages for the AI and sound modules while turning off messages for the message handler.

14.1 Public Interface

- void setOptions(const char* options)
  - parses the options string and sets debug options appropriately.
- int startLogFile(char* filename, bool append = false)
  - Prepares filename for use with the debug module
    - If filename does not exist, it creates it.
    - If append is not set to true, it will overwrite the currently existing file.
  - Returns an ID that may be used to access the file during log requests to prevent the need for string compares.
- void log(int fileID, int systemID, int Level, const char* message, ...)
  - Writes message to file with ID of fileID.
  - variable arguments will work similar to printf.
  - level and system used by debugger to filter messages
- void log(const char* filename,int systemID, int Level, const char* message, ...)
  - Works in the same manner as above but performs a string compare to determine which log file to use.
- void output(int systemID, int Level, const char* message, ...)
  - Works likes the log function except sends output to the current output location (console or standard out)specified from the options set.

14.2 Public Class Definition

```
class Debugger
{
public:
    void setOptions(const char* options);
    int startLogFile(char* filename, bool append = false);
    void log(int fileID, int systemID, int Level,
```

```
            const char* message, ...);
    void log(const char* filename,int systemID, int Level,
            const char* message, ...);
    void output(int systemID, int Level, const char* message,
                ...);
};
```

15 File Management

When saving files, it is preferable to have a common set of functions that allow saving and loading of arbitrary files while still preserving encapsulation of the data being saved and loaded. The File Memento will preserve encapsulation while still allowing the UI to choose where to save the file.

15.1 Classes

The File Memento is a virtual base class that provides the ability to be written to a std::ostream and read from a std::istream. This class is should be used solely to inherit an interface. **The File Memento is not the same as the Networking Memento!**

FileMemento	Stores the state of an object so it can be written to a file. Also is able to be created from a istream.

15.2 Dependencies

The FileMemento will be dependent on the C++ standard library's IO stream functionality in addition to being dependent on the UI to interpret it correctly.

15.3 Interfaces

```
class FileMemento
{
protected:
    FileMemento(std::istream&) {}
public:
    //FileMemento(...);
    virtual ~FileMemento(void) {}

    virtual Write(std::ostream&) const = 0;
    Read(std::istream&) = 0;
};
```

15.4 Functions

Four functions are provided by the interface FileMemento. All of these are expected to be overridden in a base class. Also, it is expected that another

constructer will be created that can fully construct the memento from data. **Do not define a default constructor**

FileMemento::FileMemento(std::istream&)
Creates a file memento from the specified stream. The memento will now have a state to be passed through the Message Handler to the specified module. The stream will be updated to have read past the data. This will normally be used for initializing a memento from the data in a file.

FileMemento::FileMemento(...)
This function does not exist in the FileMemento class. This function will be different for each derived object; its purpose is to create a memento out of currently available data in RAM.

For example, a FishMemento would most likely take a pointer to a Fish object, allowing the memento to extract all data from the Fish into the Memento.

FileMemento::~FileMemento(**void**)
Does nothing. This is a virtual function for overriding.

FileMemento::Write(std::ostream&)
Writes out the data contained in the FileMemento to the stream. This will normally be used for writting data to a file.

FileMemento::Read(std::istream&)
Reads in the data contained in the stream to the FileMemento. This will normally be used for reading data from a file.

16 Input module

The player input module will provide status of input devices used by project Aquarius. This information will be made available to the user interface. DirectInput will be used however the Input class will encapsulate its functionality.

The input module will first get the current state of all the input devices. It will then update values that have persistent components such as mouse delta positions.

16.1 Public Interface

- bool GetUserInput()
  - This method is invoked once per frame to update the state of the input devices. It returns success or failure.
- bool IsPressed(Input Constant)

  - Returns whether a key or mouse button has pressed. If GetUserInput() fails this will return false.

- bool IsTriggered(Input Constant)
  - Returns whether a key or mouse button has been triggered. If GetUserInput() fails this will return false.

- char GetLastCharacter()
  - Returns the alphanumeric character pressed during the last update. If no character was pressed this function returns 0.

- float MouseX()
  - Returns the current x position of the mouse.

- float MouseY()
  - Returns the current y position of the mouse.

- float MouseDeltaX()
  - Returns the change in the mouse's x position.

- float MouseDeltaY()
  - Returns the change in the mouse's y position.

- float MouseDeltaWheel()
  - Returns the change in the mouse wheel.

16.2 Public Class Declaration

```
class PlayerInput
{
    bool GetUserInput();
    bool IsPressed(Input Constant);
    bool IsTriggered(Input Constant);
    char GetLastCharacter();
    float MouseX();
    float MouseY();
    float MouseDeltaX();
    float MouseDeltaY();
    float MouseDeltaWheel();
};
```

16.3 Input Constants

The input constants will be remapped DirectInput Constants. Project Aquarius will not support Japanese Keyboards and thus those constants will be removed. In addition three constants have been added for the purpose of checking mouse clicks.

Key constants from DirectInput will be redefined to allow the mouse buttons to be treated the same as keyboard buttons. All constants will use the prefix DIK_, with the same suffix as in DirectInput. For example, the constant to refer to the Num Lock is KEY_NUMLOCK. The three mouse buttons will have the following constants:

Constant	Notes
KEY_LBUTTON	Left Mouse Button
KEY_RBUTTON	Right Mouse Button
KEY_WHEEL	Mouse Wheel press

17 Timer

Modules that need support for time keeping will use the timer object. It will provide functionality for checking the frame rate, time step and game time step. The game time will be a modified time step that the user will be able to adjust to speed up or slow down the speed of the game. It will have an upper and lower limit to prevent the game time being set to an unmanageable rate.

17.1 Members

- Time Sample
  - Amount of time over which to average the frame rate
- Average Frame rate
  - Average of last Time Sample amount of seconds
- Actual frame rate
  - frame rate since last frame
- Real time
  - Time step
  - Time since last frame
- Game time
  - Modifiable time step
  - Can set to constant for offline mode

- Game Speed
  - Modifier for game time
  - Limited maximum and minimum speed

17.2 Methods

- setTimeSample
  - Sets the time sample
- getAverageFrameRate
  - returns the averageFrameRate
- getActualFrameRate
  - returns the actual frame rate
- getRealTimeStep
  - returns the real time step
- getGameTimeStep
  - returns the game time step
- setGameSpeed
  - sets the game speed
- getGameSpeed
  - returns the game speed

17.3 Public Class Declaration

```
class Timing
{
public:
        void setTimeSample();
        float getAverageFrameRate();
        float getActualFrameRate();
        float getRealTimeStep();
        float getGameTimeStep();
        void setGameSpeed();
        void getGameSpeed();
};
```

18 Math

Basic linear algebra functionality is needed by multiple modules. This library will support basic linear algebra constructs, namely matrices and vectors and the ability to manipulate them. This functionality will be contained within the namespace "math".

18.1 Classes

The math library consists of the following classes. Note that each type is templated to allow use of different floating point types.

Matrix2<type>	Square Matrices of multiple sizes
Matrix3<type>	
Matrix4<type>	
Vector2<type>	Vectors of multiple sizes
Vector3<type>	
Vector4<type>	

18.2 Dependencies

The math library will be dependent on only the C++ Standard Library.

18.3 Interfaces

Each matrix class and vector class will have the same functionality, With the exception of the amount of elements it can access. Because of this, a Matrix2 and Vector2 will be the classes described, but the same basic interface will be supported by Matrix3 and Vector3.

18.3.1 Matrix2

```
template<typename type>
class Matrix2
{
public:
    //Constructors
    Matrix2(type row1col1, type row1col2,
            type row2col1, type row2col2) throw();
    Matrix2(type* MatrixData) throw();
    Matrix2(const Matrix2<type>&) throw();
    Matrix2& operator= (const Matrix2<type>&) throw();
    bool     operator==(const Vector2<type<&) throw();

    //Arithmetic operations
    Matrix2& operator+= (const Matrix2<type>&) throw();
```

```
    Matrix2& operator-= (const Matrix2<type>&) throw();
    Matrix2& operator*= (const Matrix2<type>&) throw();
    Matrix2& operator*= (type)                 throw();

    //Accessors
    //Note: operator[][] does not exist, but it will
    //appear to.
    const type& operator[][] (size_type, size_type)
        const throw();
    type& operator[][] (size_type, size_type)
        throw();
    const type* Data(void)
        const throw();

    //Special operations
    type      Det(void)       const throw();

    //Self modifying operations
    Matrix2& Transpose(void) throw();
    Matrix2& Invert(void);
};
```

18.3.2 Vector2

```
class Vector2
{
public:
    //Constructors
    Vector2(type element1, type element2) throw();
    Vector2(type* VectorData) throw();
    Vector2(const Vector2<type>&) throw();
    Vector2& operator= (const Vector2<type>&) throw();
    bool     operator==(const Vector2<type<&) throw();

    //Arithmetic operations
    Vector2 operator+= (const Vector2<type>&) throw();
    Vector2 operator-= (const Vector2<type>&) throw();
    Vector2 operator*= (const Vector2<type>&) throw();

    //Accessors
    const type& operator[] (size_type) const throw();
    type& operator[] (size_type)             throw();
    const type* Data(void);             const throw();

    //Special operations
    type Magnitude(void) const throw();
```

```
    //Self modifying operations
    Vector2& Normalize(void);
};
```

18.3.3 Shared Functions

```
//Matrix operations
Matrix2 operator+ (const Matrix2<type>&,
                   const Matrix2<type>&) throw();
Matrix2 operator- (const Matrix2<type>&,
                   const Matrix2<type>&) throw();
Matrix2 operator* (const Matrix2<type>&,
                   const Matrix2<type>&) throw();
Matrix2 operator* (type) throw();
//Vector operations
Vector2 operator+ (const Vector2<type>&,
                   const Vector2<type>&) throw();
Vector2 operator- (const Vector2<type>&,
                   const Vector2<type>&) throw();
Vector2 operator* (type) throw();
//Matrix-Vector operations
Vector2 operator* (const Matrix2&,
                   const Vector2&) throw();
Vector2 operator* (const Vector2&,
                   const Matrix2&) throw();
```

18.4 Functions

Most functions are self-descriptive, provided a knowledge of linear algebra.

Only two functions can throw exceptions, Matrix2::Invert and Vector2::Normalize. If it is not possible to execute the function (say the matrix is non-invertible), then these functions will throw the exception std::domain_error.

Also, notice that multiple functions are non-const. These functions are self modifying; Vector2::Normalize will normalize the vector and store the normalized version in the called vector.

18.5 Example code

```
void func(const Matrix2<double>&M)
{
    //calculate M inverse
    Matrix2<double> InverseM(M);
    InverseM.Invert();

    //check that M * Inverse is the identity
    Matrix2<double> Identity(1, 0,
                             0, 1);

    if(Identity == M * InverseM)
    {
        cout << "The matrices are equal!" << endl;
    }
    else
    {
        cout << "Floating point error wins again!" << endl;
    }
}
```

A Team Sign Off Sheet

Jared Finder November 6, 2002
Producer
Graphics

John Corpening November 6, 2002
Technical Director
User Interface
AI

Nate Cleveland November 6, 2002
Designer
Action Interpretation
Sound

Austin Spafford November 6, 2002
Product Manager
Physics
Environment Simulator

Nathan Frost November 6, 2002
Tester
Networking
Message Handling