INITIAL DESIGN REPORT

1395045 – Anıl Yiğit Filiz
1394642 – Berkehan Altınkaya
1394600 – Derya Akpınar
1394980 – Güneş Efe

3rd December 2006

TABLE OF CONTENTS

1. Introduction

2. Goals & Objectives

3. System Architecture
   i. DFD Level 0
   ii. DFD Level 1 for Server Game Engine
   iii. Server Game Engine
      (a) AI Module
      (b) Network Module
      (c) Message Module
      (d) Database Module
      (e) Mission Module
      (f) Visible Area Module
   iv. DFD Level 1 for Client Game Engine
   v. Client Game Engine
      (a) Network Module
      (b) Graphics Module
      (c) Sound Module
      (d) Physics Module

4. Sequence Diagrams

5. E-R Diagram

6. Graphical User Interfaces
   i. Description of User Interfaces
   ii. Use-Case Diagram
   iii. State Transition Diagram

7. Classes
   i. Class Diagram
   ii. Class Definitions

8. Testing Issues
   i. Test Design
   ii. Test Cases

9. Appendix
   i. Model Spaceships
   ii. Gantt Chart

1. Introduction

Massively multiplayer online games is the new trend in gaming. With the aim of creating an exciting and consistent universe, our team is planning to design a space simulation game that has a science-fiction scenario. Like every traditional MMOG our game will have registered users and there will be experience based level-ups while playing the game. The storyline of Twilight is as follows:

"In the year 2525 the prophecy of the Goths (a religious human minority living close to the planet Pharma) came true and the galaxy Quasar appeared in the middle of the universe. The new active galaxy Quasar is the home of the biological beings named Eaons. Goths proclaimed that the appearance of the Eaons was the last sign of the apocalypse of the universe. Goths built an army to wipe out Eaons once and for all. But there was one thing that was not foreseen. In the far planets of Quasar there was another race named Cyrians which depended upon X-Particles that only Eaons could produce. The universe was being led to chaos. Darkness to brightness or brightness to darkness. Choose between Humans, Goths and Cyrians while the apocalypse arrives."

There will be three races in Twilight namely Humans, Goths and Cyrians.

Goth race is a rebellion of Humans whom are full of anger and has no mercy against the other livings around them. This anger, in time led them to discover their own religion and life-style. Their priests have their own perspectives about the apocalypse day (which in fact inherited from human religions) and this foreseen perspective came true with the appearance of the galaxy Quasar. So the Goths are enemies of both Humans and Eaons. This will then lead to a diplomacy change with Cyrians who depends on the Eaons. Goth ships will have more attack power than other races.

Humans are neutral against Cyrians. But they are enemies of Goths. But when attacked by a Cyrian the diplomacy will change. Humans ships are balanced. They have normal attack and defense powers.

Cyrians are the protectors of Eaons. Since they depend on the X-Particles produced by

Eaons, when Humans or Goths attack any Eaon, the temporary diplomacy will change. Since Eaons supply the necessary X-Particles to Eaons it is a matter of death to protect Eaons for Cyrians. Cyrian ships will have more defensive power than other races.

2. Goals and Objectives

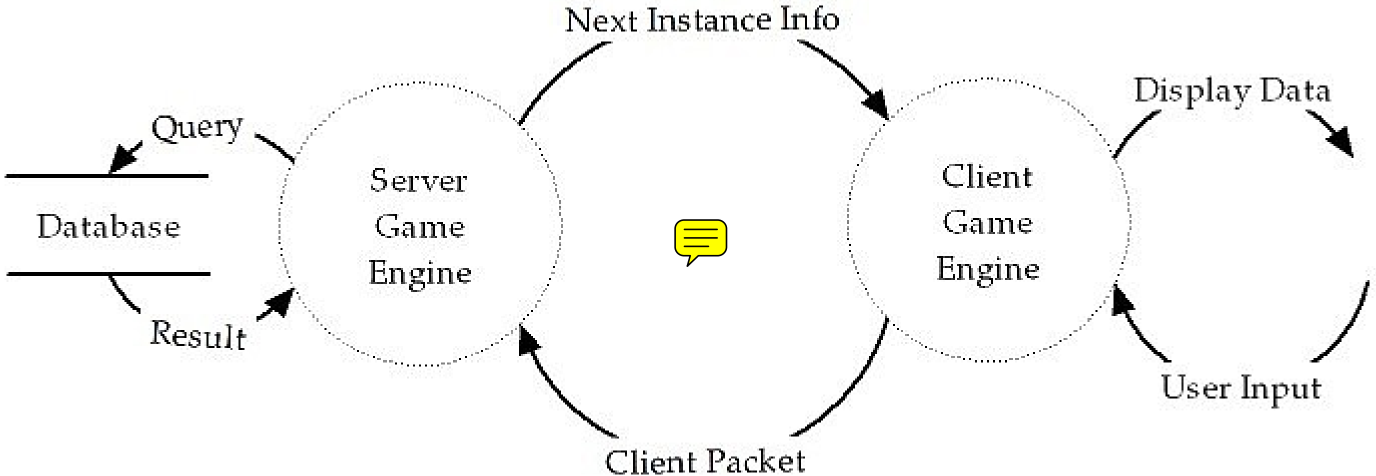
During the design phase of we had a lot of discussion about the various aspects of the game. Our primary objective is to create something which is small in size and great in efficiency. Our primary objectives are as follows:

- ✔ By the end of this term we should be finished with the network modules of our game. The packet size and the thread management issues inside the send and receive methods are very important issues since this game is an online multiplayer game. So we spent and will spend fair amount of time for the networking. We decided to use WinSock API for packet transfers via TCP/IP.
- ✔ Determining the states of the game is another important issue. Since games are nearly state machines which does something different by looking at the states we are thinking of writing our own state machine. But in the state design there will also be sub-states. For example in the "User Interface" state there will be a sub-state named "Buy". This state, sub-state coordination will make it easier to handle the rendering of the screen.
- ✔ Another must of a game is using effective sound effects. Since we'll implement this game using c++ we should find a suitable API for playing and manipulating sound files. There were some candidates for the API choice (OpenAL, SDL, DirectSound) but we agreed on using DirectSound which is the sound API of DirectX SDK. So learning about this API is another thing to do.
- ✔ The user information will be held in a database in the server. So in a little part of our game we will be concerned with connecting to this database and retrieving and storing information on this database. So a suitable API for connecting to this database will be needed. We will use MySQL on our server. A suitable API to connect to this database is MySQL++ and we should also learn about that API.
- ✔ There will be lots of 3D objects in the game. Loading this 3D objects and placing textures to their surfaces is another thing to do. We'll use Object3DS for that purpose and we should also learn about that API. Since we also got a parallel graphics project including the usage of

this API we'll be learning about Object3DS soon. We've started modeling some of our spaceships. These can be seen in the appendix.

3. Architectural & Component Level Design

i. DFD Level 0



ii. DFD Level 1 for Server Game Engine

Encoded Message
Client Packet
User State
Query
Network Module
Message Module
Database Module
DBMS
Result
Mission Info
Next Instance Info
User Info
Client Data
User Data Update
Mission Module
Visible Area Module
NPC Data Update
Auto Trigger
AI Module
Game Data
User Initial Data
Client Initializer

iii. Server Game Engine

Server game engine consists of a main game loop which runs a couple of modules in an order to supply the sequentiality of the game. This main game loop accepts the packets from the clients. Regarding to the messages received from these clients it triggers other modules to do their stuff. The modules and their features are listed below.

(a) Network Module

Network module is responsible for the message traffic. It will be a multi-threaded network program which listens to a specified port on the server. When a user makes a connection request to the port listened by the server, network module creates another thread which in turn opens another socket for that client to fulfill its own send and receive operations. A socket cannot send and receive at the same time. But this is not a problem in our case since we are planning to send the next instance info if and only if a packet received by a client. So at first the received packet will be processed, and then the information for the next time instance will be calculated and sent to the clients. The state of the socket assigned to the user will change between send and listen modes. And this change will be exactly the inverse of the client. Which means if the server thread is in send mode, the user assigned to that thread will be in listen mode. Packet descriptions are as follows:

Client Packets		
Packet Header	**Packet Content**	**Packet Description**
Invalid	-	When the packet is invalid the connection between the server and client is broken and the server assumes that the user has sent a quit message.
Register_User	int username_len + string username + int password_len + string password + char race	The packet contains the user's desired username and password and the race he/she selected.
Login_User	int username_len + string username + int password_len + string password	The packet contains the user's claim of his/her username and password.
Quit_User	-	The packet is sent when the user informs the server that he/she has quit. It contains no information since the user's id exists in the ip to id table stored in the server.
Enter_Hangar	int ship_id	The packet contains the id of the ship that the user entered the hangar with.
Exit_Hangar	int ship_id	The packet contains the id of the ship that the user has selected to leave the hangar with.

Mission_Accept	int mission_id	The packet contains the id of the mission that the user accepted in the mission select screen.
Buy_Item	int ship_id + char object_type	The packet contains the id of the ship that the user has bought an item on and the type of the item. If the user has bought a ship the ship id is 0 and the object type is the type of the ship.
Sell_Item	int ship_id + char object_type	The packet contains the id of the ship that the user has sold an item on and the type of the item. If the user has bought a ship the ship id is 0 and the object type is the type of the ship.
Game_Data	arbitrary number of objects	The packet contains all of the user's game data parsed as different objects and concatenated one after another. These objects can be in any order and are of previously defined sizes.

Server Packets		
Packet Header	**Packet Content**	**Packet Description**
Invalid	-	When the packet is invalid the connection between the server and client is broken and the server assumes that the user has sent a quit message.
Invalid_Login	-	The packet is sent to inform the user that the received username and password do not match the username and password in the database.
Invalid_Username	-	The packet is sent to inform the user that the desired username is not available.
Hangar_Data	the object data types of the three ships that the user has in the database + the amounts and prices of all of the items offered in the hangar + possible sell prices of all of the items offered in hangar + the object data types of arbitrary number of available missions in the hangar	The packet contains all of the information that is needed to display the hangar screen. The buy and sell prices of all items and the properties of the user's ships are sent as objects concatenated one after another.
Game_Data	arbitrary number of objects	The packet contains all of the user related game data parsed as different objects and concatenated one after another. These objects can be in any order and are of previously defined sizes.

Object Packets		
Object Header	**Object Content**	**Object Description**
Effect	long unsigned int timestamp + int owner_id + int effected_id + char effect_type + float effect_amount	The effect (rocket hit, laser hit, magnet pull, touch (for some missions), vs.) of one object to another.
Spaceship	long unsigned int timestamp + int owner_id + int ship_id + char ship_type + int owner_experience + float x + float y + float z + float Rx + float Ry + float Rz + float health + float shield + int body_level + char body_state + char body_frame + int laser_level + char laser_state + char laser_frame + int rocket_level + char rocket_state + char rocket_frame + int turret_level + char turret_state + char turret_frame + int mine_level + char mine_state + char mine_frame + int magnet_level + char magnet_state + char magnet_frame + int stealth_level + char stealth_state + char stealth_frame + char selected_weapon + int rocket_amount + float rocket_health + int mine_amount + float mine_health + float laser_health + float turret_health + int magnet_amount + float	The spaceship of a user. Also contains the user's current game info since a user can have only one ship while the game is running.

	magnet_health + int stealth_amount + float stealth_health	
Rocket	long unsigned int timestamp + int owner_id + int rocket_id + int rocket_life + char rocket_level + char rocket_state + char rocket_frame + float x + float y + float z + float Rx + float Ry + float Rz	The rocket of a user flying in space.
Laser	long unsigned int timestamp + int owner_id + int laser_id + int laser_life + char laser_state + char laser_level + char laser_frame + float x + float y + float z + float Rx + float Ry + float Rz	The laser of a user flying in space.
Mine	long unsigned int timestamp + int owner_id + int mine_id + int mine_life + char mine_state + char mine_level + char mine_frame + float x + float y + float z + float Rx + float Ry + float Rz	The mine of a user flying in space.
Mission	long unsigned int timestamp + int owner_id + int mission_id + char mission_type + char mission_state	Current mission data of the user.
Chat	long unsigned int timestamp + int chat_len + string chat	Recently sent chat of the user.

Packet to be sent in network messaging of the game :

- All the packets will start with a timestamp.

This timestamp will be useful for the following issues:

- There will be an optimization in the server side in the decision of the information to be sent to each user. If the timestamp of the object in the game environment are less than the timestamp of the packet received by the user, server doesn't have to send these objects to the client since the client has already got the information of that object. Server will only send information of the objects that has a bigger timestamp than the packet received by the user.
- When a packet received by a user has a field regarding to the object, the decision of updating the object will be up to the timestamps. If the packet has a bigger timestamp then that object needs to be updated and its timestamp must be "*timestamp+1*". If the packet has a smaller timestamp this means that the packets arrived to the server belongs to an old action of the client, which is probably based on a lag in the network. The old information in the packets will simply be ignored.

- After the timestamp there will be the information about the objects in the game environment. These subfields in the packet will have the convention <H><I>, where <H> is the header of the object and <I> is the information about that object. Information contents are shown in the tables above.

(b) Message Module

The packets received by the network module will be passed to a message module. This message module will be needed to decode the received messages and set specific states for the game engine to trigger different modules. This specific states can be listed as login, register, quit, mission requests. This message module is responsible for encoding the messages to be sent to the clients.

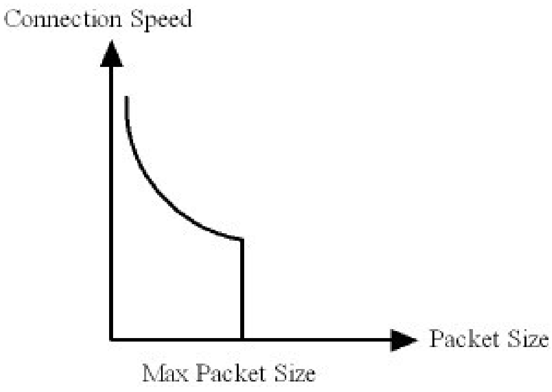
(c) Database Module

Since the client informations will be held at a database there is also a need for a database module. This database module will be active if a user wants to register, login or quit. Other than these user requests also a timeout of the connection of a user will trigger database module. Database module will do the following in the conditions below:

- ✔ Register: Look if the username exists, if not insert a new tuple in the database.
- ✔ Login: Check if the username and password tuple is in the database.
- ✔ Quit & Timeout: Update the last user information on the database. That is, the features of the users ship and the user experience.

(d) Visible Area Module

Visible area module is a very important pat of the server. This module will calculate the visible area of the user and only send those changed object information within that visible area. This is a very good optimization for the packet size to be smaller since the clients will only get the environment information thats only changed at every time packet receive. This means that faster the connection of the user, smaller the packet size will be. This can be shown with the aid of the following graph.



(e) Mission Module

Missions are an important part of the game since, in a static universe players can get bored easily. The missions will be delivered in the HQ of the players in their hangars. This process will be as follows:

- ✔ The mission module will return a set of available missions to the user when he/she enters the his hangar.
- ✔ When a user accepts a mission the mission will be initialized by the mission module, that is the objects regarding to that mission will be updated to the game data.
- ✔ After that initialization phase the mission module will be responsible for tracing the mission objectives. It will also decide if the mission is accomplished or failed.

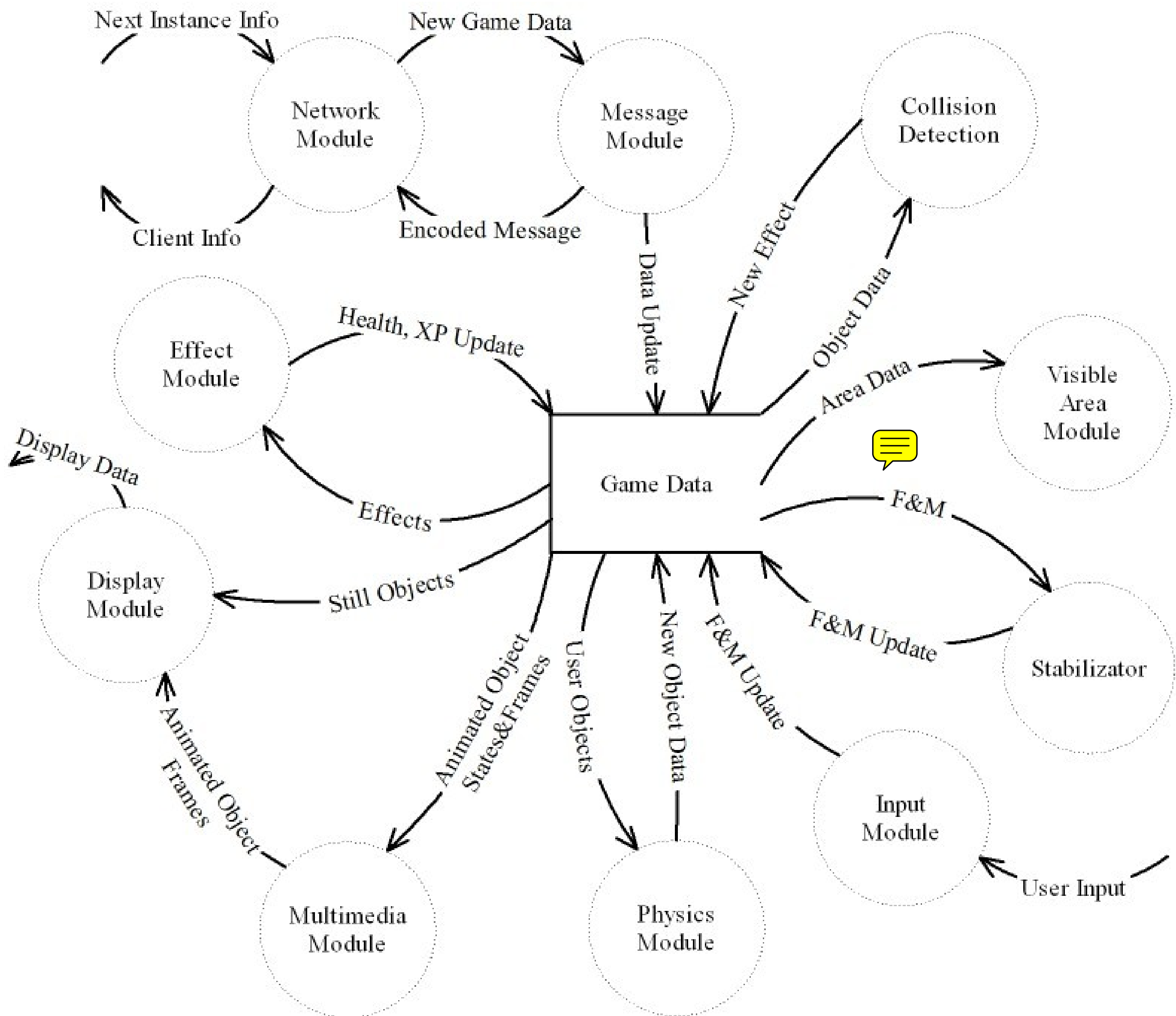
(f) AI Module

The NPC's in the game will be controlled by this AI module. The AI module will access to the game data and update the NPC information in a specified interval. The NPC's are Eaons in our game. There will be 5 different types of Eaons. This types are sorted with increasing attack power and defense below:

Sibling –> Breeder –> Psychic –> Hunter –> Mother

These order also defines the increasing intelligence order of the NPCs besides the attack & defense power. The NPCs will also level-up within the game and this level-up will result in a restoration of the health of the NPCs.

iv. DFD for Client Game Engine



v. Client Game Engine

(a) Network Module

This network module has a single thread and manages the send and receive commands one after another. There will be a single socket connected to the game server and this socket will be initiated to send mode at first. After sending a message to the server it will jump into receive state and wait for a message from the server.

(b) Message Module

The message module on the client is the same as the message module of the server. Since client and server uses the same language(that is the packet information in our case), the

translator(that is message module in our case) is the same for both of the sides. This module will be responsible for encoding and decoding the messages. When decoded an update to the game data will be done. When encoded the packet to be sent to user will be passed to the Network Module.

(c) Collision Detection

Collision detection will simply detect the collisions to the user. If anything crashes to the user at that time instance, a new effect will be appended to the game data to be processed by the effect module.

(d) Effect Module

Effect module will process the effect objects in the game data. This process will result in either health and experience update of the user or a change in the object state. The health and experience updates will be checked for an event of death or level-up. The animations regarding to these game states will be initiated for the Multimedia Module to handle them.

(e) Visible Area Module

Visible Area Module will be responsible for cleaning up the old game data received from the server as well as extracting the objects from the game data which are visible in the game-play screen.

(f) Input Module

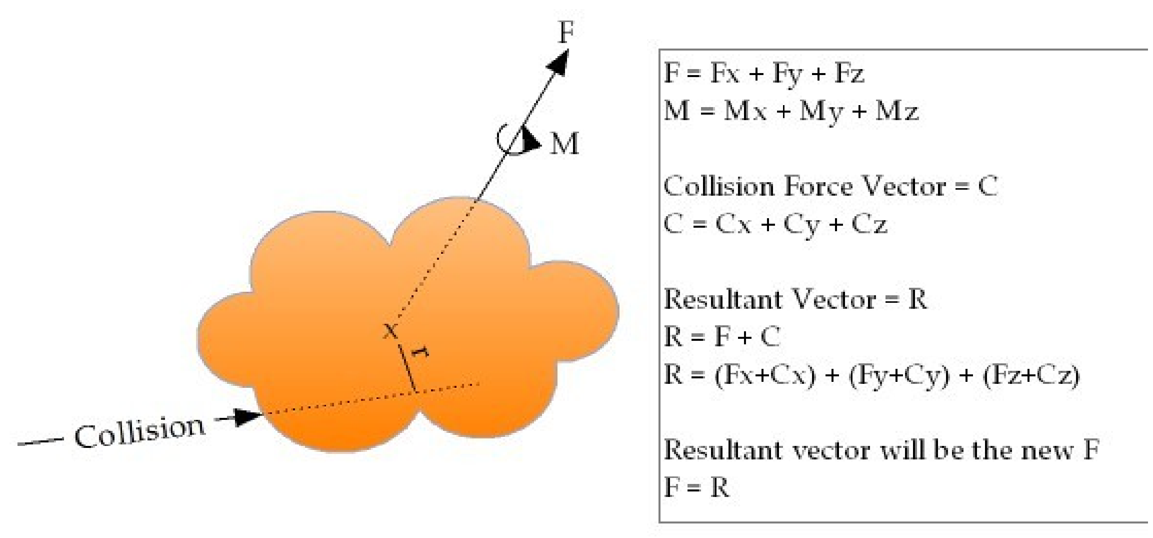
Input Module will be responsible for handling the inputs received from the user via keyboard or mouse. For the mouse clicks it will queue the user input to be processed when a new packet from the server arrives. In every loop it will update the *force* and the *moment* vectors of the user with respect to the input received from mouse and keyboard.

(g) Stabilizer

The stabilizer will be responsible for stabilizing the user spaceship. It will try to balance the *force* and *moment* vectors. This module will act like a booster to the opposite direction of the movement of the ship.

(h) Physics Module

Physics module will calculate the next instance of the user objects with respect to their *force* and *moment* vectors. This module will fetch all the external force vectors applying to user objects and calculate the new *force* and *moment* vectors. The simple logic is shown in the diagram below.



r is the position vector which is perpendicular to the center of gravity of the object.

M = F x r   where x is cross product.

Mx = Mx + Mcx   where Mc = C x r
My = My + Mcy
Mz = Mz + Mcz

At every instance the x, y and z components of F and M will be calculated.

From the force vector the acceleration will be calculated and the coordinates of the next instance will be determined. This can be formulated as follows;

$F_x = m \cdot a_x$ $F_y = m \cdot a_y$ $F_z = m \cdot a_z$

$\Delta x = \frac{1}{2} \cdot a_x \cdot t^2$ $\Delta y = \frac{1}{2} \cdot a_y \cdot t^2$ $\Delta z = \frac{1}{2} \cdot a_z \cdot t^2$

The same approach can be used for the rotation amounts with respect to x, y and z coordinates. Using the rotation vectors M_x, M_y and M_z the angle that one object has with respect to the x, y, z coordinates can be formulated as follows;

$r_x = (M_x / c) * 360$ $r_y = (M_y / c) * 360$ $r_z = (M_z / c) * 360$

where c is a constant for determining a fair amount of degree to the coordinates which must

be a larger number than M values.

(i) Multimedia Module

Multimedia Module will handle both animation and sound events. Looking at the state of one object it will send the regarding frame of that object to display module and increment the frame by one at each loop. If there are no frames left to display in that state or a terminating condition occurs to switch into another state it will also handle these situations too. At the very beginning of animations it may call callback functions to produce sound or do any necessary actions. For example when a user fires his/her laser the laser will not appear in the screen at that instant. First an introductory animation will be played and then the laser itself will be created. These are two different states which this module must have control over. While playing the introductory animation for the laser an appropriate sound must be played as well. And it must last for only the time interval of that animation.

(j) Display Module

Display Module is responsible for drawing everything that is visible to the screen. All the drawable objects in the game data will be displayed with respect to its state and frame number. These states and frame numbers will be controlled by the multimedia module. The only thing display does is to draw these objects to the right places on the screen. This module will include loading the objects(3ds), textures as well as making the appropriate adjustments of light, color and blending for the user interfaces to seem in a more fashionable way.

4. Sequence Diagrams

Game Engine
Network Module
Network Receive Module
Network Decode Module
listen()
receive()
decode()

Sequence Diagram

Network Decode
Database Module
Spawn Point
Visible Area Module
Encode-Send Module
process_database()
calculate_spawn_pt()
calculate_vis_area()
encode()

Sequence Diagram

Network Decode

Database Module

Encode Module

Send Module

Clean Game Data

Clean Mission

process_database()

encode()

send()

Quit / Timeout + Flush DB

clean()

Sequence Diagram

Network Decode

Mission Module

Visible Area Module

Encode Module

Send Module

init_mission()

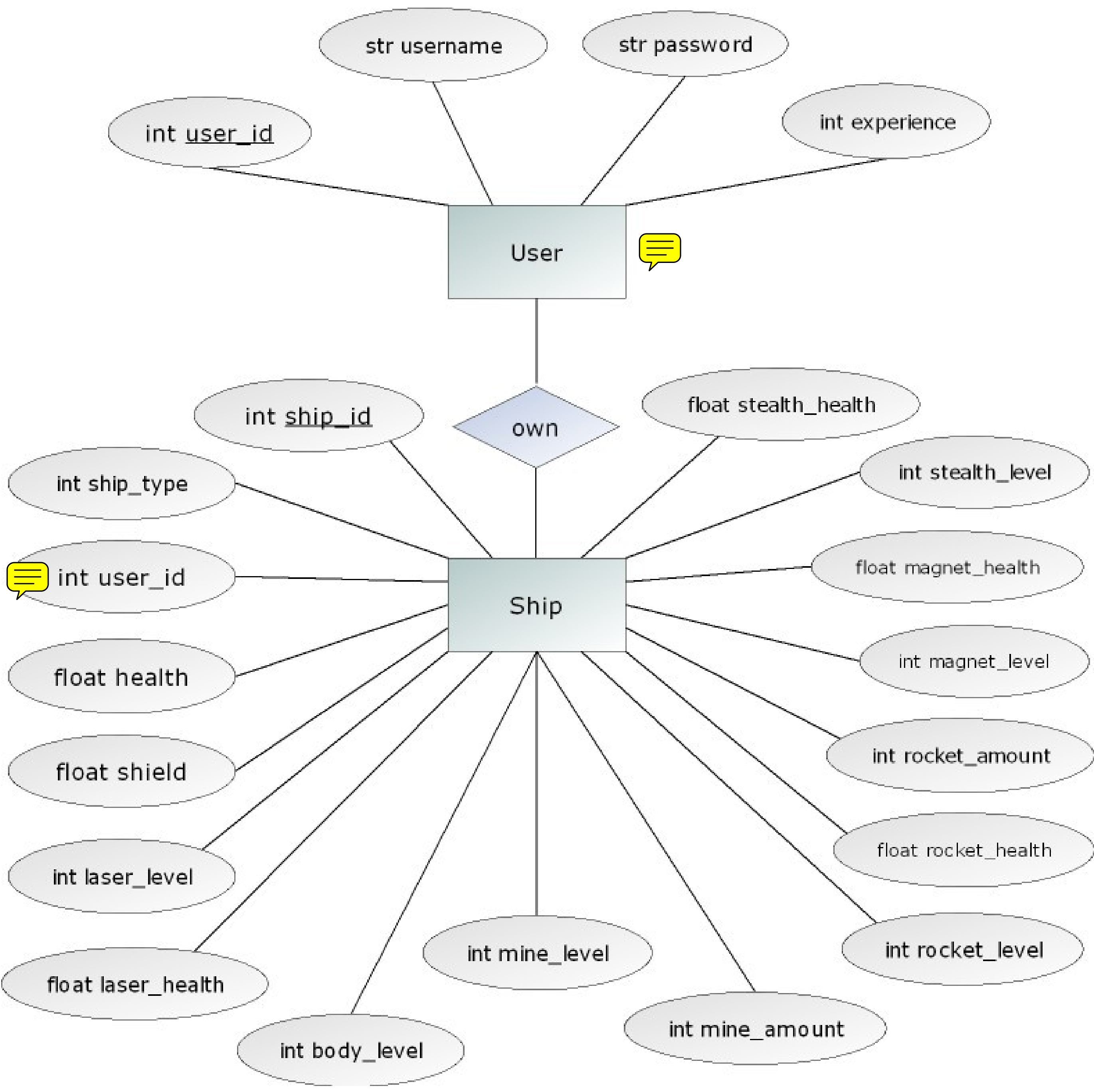
calculate_vis_area()

encode()

send()

Sequence Diagram

5. E-R Diagram



In our database there are two entities. User entities has the primary key user_id which means every user has a unique id. Second entity is ship with the primary key ship_id. Ship entity have a foreign key user_id which gives the owner of the ship. These two entities have one-to-many relation between each other. A user can own more than one ship in the database.

6. Graphical User Interfaces

i. Description of User Interfaces

The interfaces shown to the user will be ordered as follows:

- ➔ Intro Animations
- ➔ Register / Login Screen
  - Race Selection Screen
  - Register Screen
- ➔ Player Hangar Screen
  - Buy Screen
  - Sell Screen
  - Mission Screen
- ➔ Loading Animation
- ➔ Gameplay Screen

The introduction animation of our company Dirty Pixel is the first screen in the game, then another animation of Twilight will be shown to the user which includes the storyline and scenes from the game.

The third screen is login screen for registered players which has a link to the registration screen for unregistered users. In that screen registered users should choose a server IP and a server port after entering username and password for login. The login button logs the user to the game. Exit button is the exit point of the program when clicked the program terminates. After logging into the system registered users directed to the loading screen. Register button directs the system to the registration page.

Unregistered users pass to the registration screen where first a race selection screen welcomes the player. A new user account created according to a unique username. Create account button saves the username and password to the database of the server.

After clicking the register button user is directed to the race selection screen. In this screen

information about races will be given. On click of any of these races means a selection and the username & password screen will then displayed.

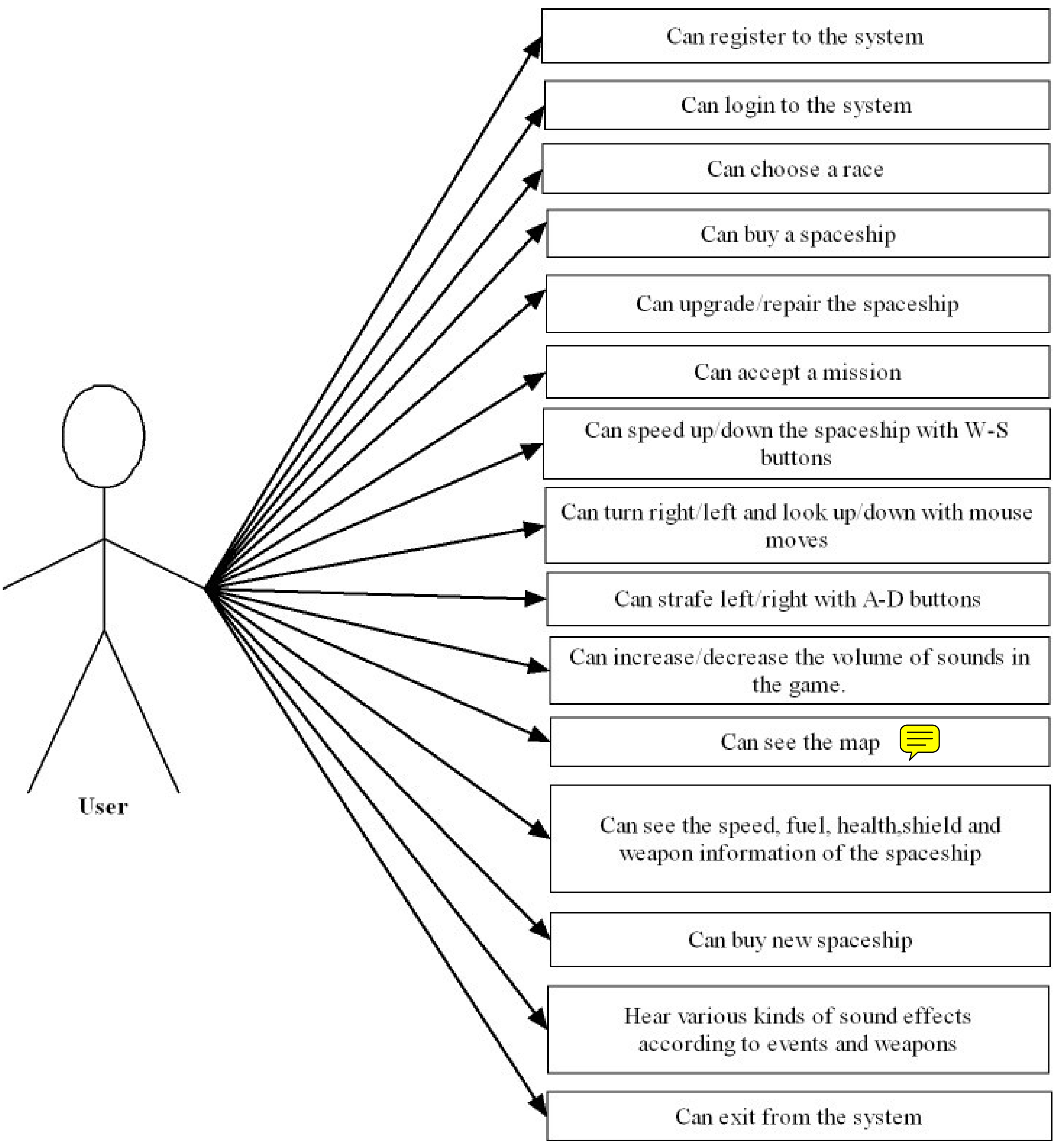
The next step after choosing the race and username is buying the first spaceship for starting the game. User chooses the spaceship according to the informations about the ships in the spaceship buy page but he/she can buy a spaceship only if his/her money is enough for the selected ship. The spaceship saved to the player account in the database when the buy button clicked. Game is ready for loading for new user after selecting the spaceship.

Users spawn in their hangars which is in their HQ where they can repair or upgrade their spaceship in exchange to their money. Buy spaceship screen opens when users want to repair and upgrade their spaceship or buy a new spaceship by clicking the related place in the headquarter. Sell spaceship screen opens when users need to sell spaceship to earn money for a specific purpose.

The special missions are also assigned in the headquarters of the users by the mission screen. Mission screen appears when the user clicks the computer icon in the headquarter. User is ready to play the game after login, settings and accepting the mission.

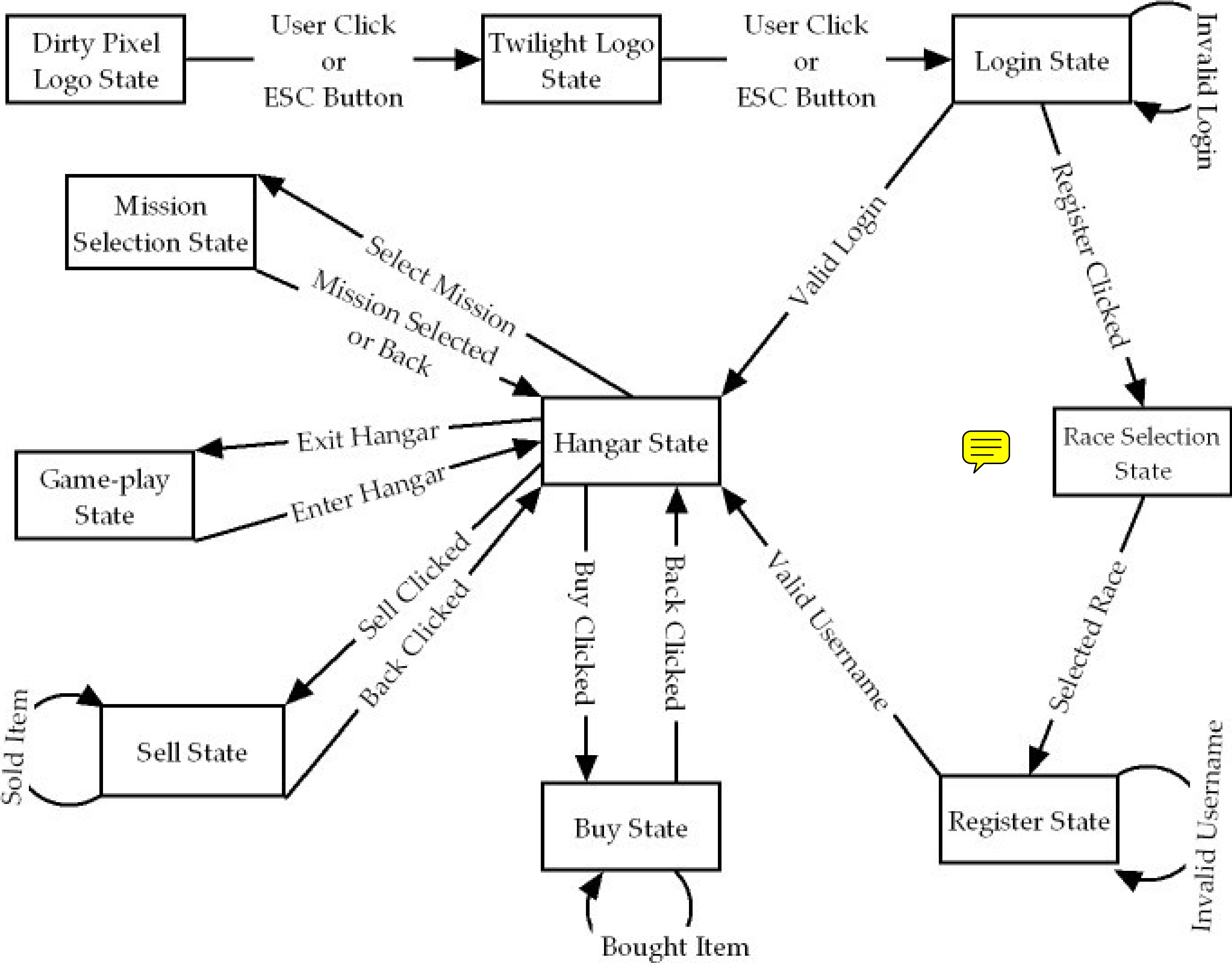
The game play screen shows all needed informations to the player. In the left side of the screen the info of shield amount is given, on the right side the health info is given vertically. In the bottom left side of the screen the map of the universe is placed. Player can see the fuel and speed info in the bottom-center part of the game play screen. To the left of the fuel info the weapon type and the amount of the shots is given. To the right of the speed info, the type and amount of the mines are given. The chat screen is placed in the bottom-right part of the screen. Information screens are designed in a compact way for giving a larger view to the player in game environment. The sketches for the user interfaces can be found in the appendix.

ii. Use-Case Diagram



The possible actions of the user is shown above. The actions are given in order. User interface actions are followed by the game-play actions.

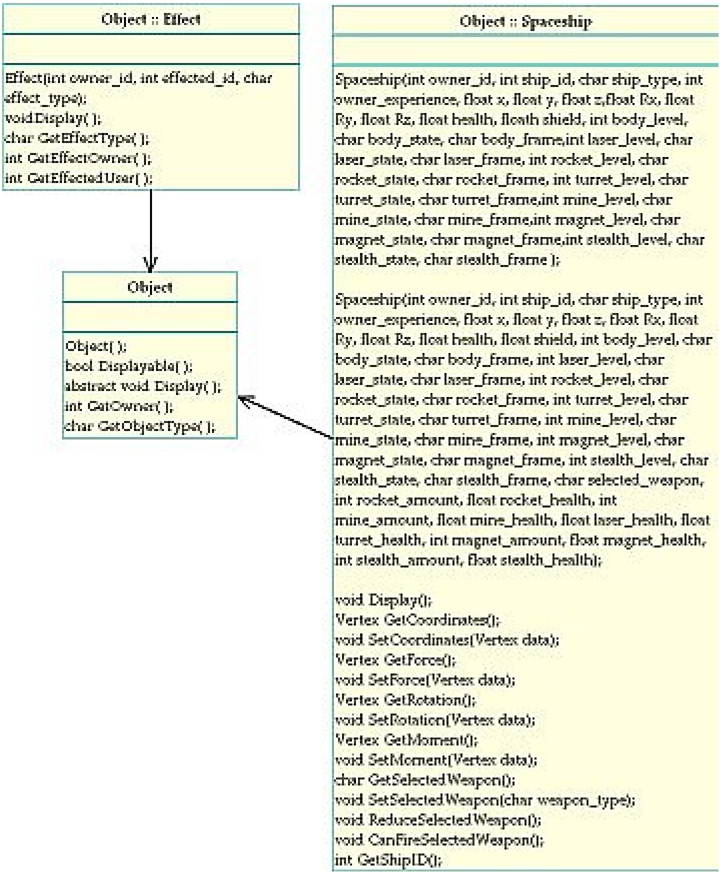
iii. State Transition Diagram

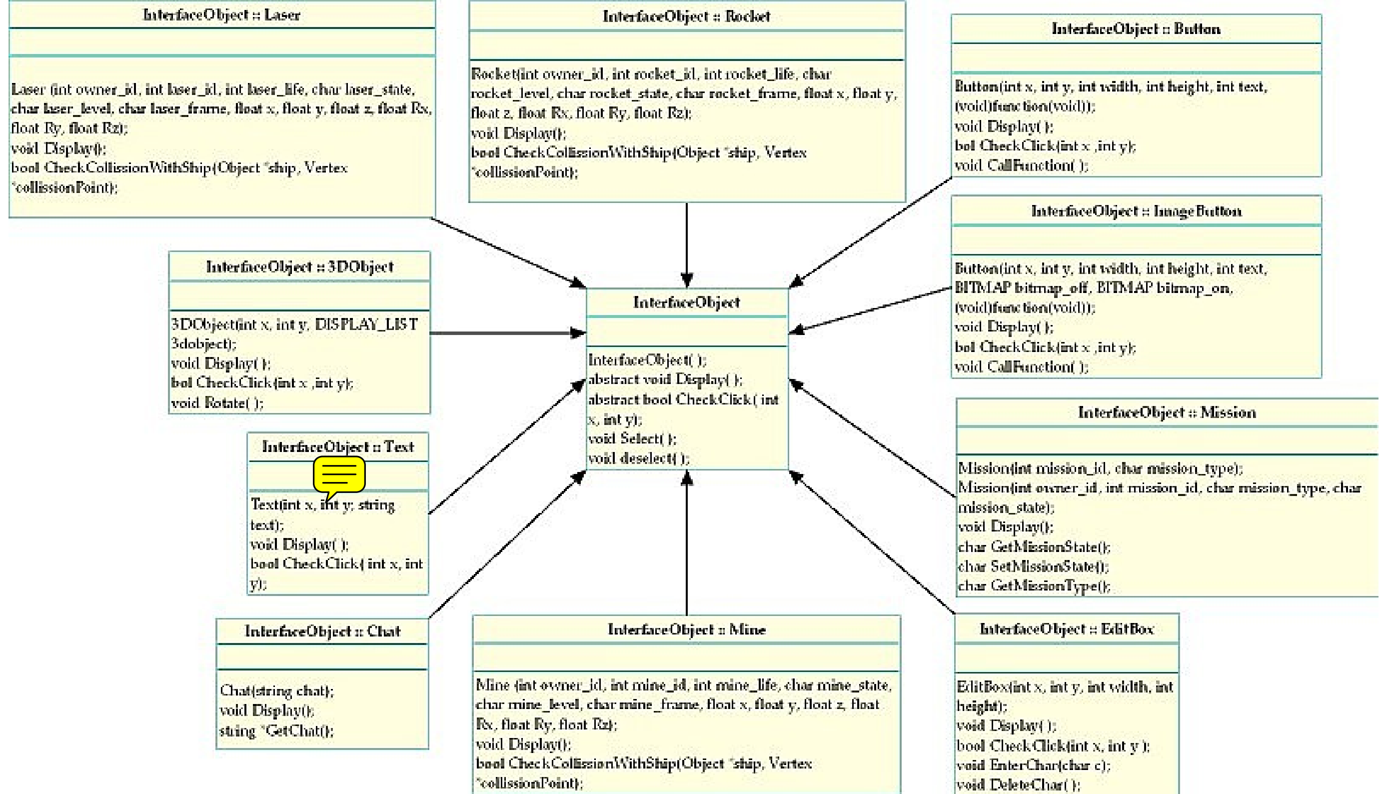


This state transition diagram explains the flow of the game from the start. It's actually explained above. These states will activate the regarding modules. But it gives an insight of the actual flow of the game.

7. Classes

i. Class Diagrams





Vertex

float X( );
float Y( );
float Z( );
Vertex(float x, float y, float z);

DecodeModule

DecodeModule( );
StartDecode(string *data);
char GetHeader( );
int GetInt( );
string &GetString( );
char GetChar( );
float GetFloat( );
Object *GetMyShip( );
Object *GetObject( );

EncodeModule

EncodeModule( );
StartEncode( );
void PutHeader(char data);
void PutInt(int data);
void PutString(string data);
void PutChar(char data);
void PutFload(float data);
void PutMyShip(object *data);
void PutObject(data);
string *GetBuffer( );

GameData

GameData( );
void UpdateObject(object *obj);
void UpdateMyShip(object *obj);
void SelectFirstObject( );
object GetObject( );
void SelectMyFirstObject(int user_id);
Object *GetMyFirstObject( );
Spaceship *MyShip( );
void DeleteDistantObjects (int user_id);
Object ReturnMyPerimeter(int user_id);

Multimedia Module

MultimediaModule( );
void HandleSoundsOfAllObjects(GameData *data, int user_id);

NetworkServer

NetworkServer(int port);
string *GetReceiveData( );
bool SendData(string *data);

Camera

Camera(GameData *data);
void MoveCameraToItsPlace(GameData *data, int user_id);

NetworkClient

NetworkClient(int port, string ip);
string *GetReceiceData( );
bool SendData(string *data);

MissionModule

MissionModule( );
void PrepareNewMissions( );
void CheckMissionStates(GameData *data, int user_id);
void UserAcceptMission(GameData *data, int user_id, int mission_id);

EffectsProcessorModule

void HandleEffects(GameData *data,int user_id);
void HandleExperience(GameData *data, int user_id);
void HandleMyObjects(GameData *data,int user_id);

PhysicsModule

PhysicsModule( );
void MoveMyObtects(GameData *data, int user_id);
void CheckMyCoordinates(GameData *data, int user_id);
void StabilizeMyShip(GameData *data, int user_id);

DatabaseModule

DatabaseModule(string username, string password, string host);
bool CheckValidUsername(string username);
bool CheckValidUsernameAndPassword (string Username, string password);
int InsertNewUser(string username, string password, string race);
void UserEnteredShipToHangar (GameData *data, int user_id, int ship_id);
void UserBoughtItem (int user_id, int ship_id, char object_id);
void UserSoldItem (int user_id, int ship_id,char object_id);

Timer

Timer ( );
void HandleAllTimerVariables( );
void AddTimerVariable(int *variable, float start_value, float end_value, int time_interval char func_type, (void*)function(void));

InputHandler

InputHandler( );
static void KeyboardUp(char key, int x, int y);
static void KeyboardDown(char key, int x, int y);
static void SpecialKeyboardUp(char key, int x, int y);
static void SpecialKeyboardDown(char key, int x, int y);
static void PassiveMouseMove(int x, int y);
static void MouseMove(int x, int y);
static void MouseClick(int button, ,nt button_state, int x, int y);
void HandleKeyboardInput(GameData *data, int user_id);
void HandleMouseInput(GameData *data, int user_id);

ServerLoop

ServerLoop();
void MainLoop();

ClientLoop

ClientLoop();
void MainLoop();
int DirtyPixelLogoLoop();
int TwilightLogoLoop();
int LoginScreenLoop();
int RegisterScreenLoop();
int LoadingScreenLoop();
int RaceSelectionScreenLoop();
int HangarScreenLoop();
int BuyScreenLoop();
int SellScreenLoop();
int MissionSelectionScreenLoop();
int GameplayLoop();

AIModule

AIModule();
void CreateNewNpcs(GameData *data);
void HandleNpcEffects(GameData *data);
void DetermineNpcsState(GameData *data);
void HandleNpcs(GameData *data);

CollisionDetection

CollisionDetection( );
void DetectAllCollissions(GameData *data,int user_id);

ChatBox

ChatBox(GameData *data, int x, int y);
void Display();
void SendChat();
void EnterChar(char c);
void DeleteChar();

FuelBox

FuelBox(GameData *data, int user_id, int x, int y);
void Display();

InformationBox

InformationBox(int x, int y);
void PutInformation(string info, int time, int priority);
void Display();

EnemyDetails

EnemyDetails(GameData *data, int user_id);
int FindSelectedShip();
bool Locked();
void Display();

UserInterface

UserInterface(BITMAP background);
void Activate( );
void Display( );
void AddInterfaceObject(Interface Object *obj);
static void KeyboardUp(char key, int x, int y);
static void KeyboardDown(char key, int x, int y);
static void SpecialKeyboardUp(char key, int x, int y);
static void SpecialKeyboardDown(char key, int x, int y);
static void PassiveMouseMove(int x, int y);
static void MouseMove(int x, int y);
static void MouseClick(int button, ,nt button_state, int x, int y);

MapBox

MapBox(GameData *data, int user_id, int x, int y);
void Display();

ExperienceBox

ExperienceBox(GameData *data, int user_id, int x, int y);
void Display();

SelectedWeaponBox

SelectedWeaponBox(GameData *data, int user_id, int x, int y);
void Display();

SpeedBox

SpeedBox(GameData *data, int user_id, int x, int y);
void Display();

HealthBox

HealthBox(GameData *data, int user_id, int x, int y);
void Display();

ShieldBox

ShieldBox(GameData *data, int user_id, int x, int y);
void Display();

ii. Class Definitions

```
class Vertex {

        Vertex(float x, float y, float z);
        /*      Initializes the vertex.
        */
        float X();
        float Y();
        float Z();
}

class Object {
        Object();
        /*      Initializes an object with default values.
        */
        bool Displayable();
        /*      Returns true if the object is a solid object with x,y and z coordinates and
                should be drawn to the screen.
        */
        abstract void Display();
        /*      Draws the object by calling the corresponding display function.
        */
        int GetOwner();
        /*      Returns the owner of the object.
        */
        char GetObjectType();
        /*      Returns the type of the object. It is either a ship, a particle , a mission or an
                effect.
        */
        intGetTimestamp();
        /* Returns the time-stamp of the object.
        */
}

class NetworkServer {
        NetworkServer(int port);
        /*      Initiates a thread to receive new connections and control their receives on
                the specified port.
        */
        string *GetReceivedData();
        /*      Returns the latest received data from anyone of the connected clients. The
                client's thread gets ready to send message. Returns NULL if no data was
                received.
        */
        bool SendData(string *data);
        /*      Sends a data to the client that was received from. Cannot be called if no
                receive was made.
```

```
	*/
}

class NetworkClient {
	NetworkClient(int port, string ip);
	/*	Initiates a thread to make a connection to the specified server. The thread
		then gets ready to send message.
	*/
	string *GetReceivedData();
	/*	Returns the latest received data from the server. Returns NULL if no data
		was received. Cannot be called before a send is made.
	*/
	bool SendData(string *data);
	/*	Sends the data to the client. The thread then waits for a receive.
	*/
}

class DecodeModule {
	DecodeModule();
	/*	Initializes the decode module class.
	*/
	StartDecode(string *data);
	/*	Initializes the pointer to the first object on the data.
	*/
	char GetHeader();
	/*	Gets the header of the received data. Moves the pointer to next data.
	*/
	int GetInt();
	/*	Gets an integer from the received data. Moves the pointer to next data.
	*/
	string &GetString();
	/*	Gets an integer from the received data. Then gets a string the length of the
		integer received. Moves the pointer to next data.
	*/
	char GetChar();
	/*	Gets a char from the received data. Moves the pointer to next data.
	*/
	float GetFloat();
	/*	Gets a float from the received data. Moves the pointer to next data.
	*/
	Object *GetObject();
	/*	Gets an object from the received data and allocates a space for it by looking at the
		type of the object and initializing it's corresponing class with the received values. The
		pointer is incremented to the next object. Returns NULL if the end of the data is
		reached.
	*/
}
```

```
class EncodeModule {
	EncodeModule ();
	/*	Initializes the encode module class.
	*/
	StartEncode();
	/*	Deletes the previous buffer if it exists. Makes the byte count zero.
	*/
	void PutHeader(char data);
	/*	Appends the header to the buffer.
	*/
	void PutInt(int data);
	/*	Appends an integer to the buffer.
	*/
	void PutString(string data);
	/*	Appends the length of the string to the buffer. Then appends the string to
		the buffer.
	*/
	void PutChar(char data);
	/*	Appends a char to the buffer.
	*/
	void PutFloat(float data);
	/*	Appends a float to the buffer.
	*/
	void PutObject(data);
	/*	Encodes and appends an object to the buffer.
	*/
	string *GetBuffer();
	/*	Gets the pointer of the encoded data.
	*/
}
```

```
class DatabaseModule {
	DatabaseModule(string username, string password, string host);
	/*	Creates a connection to the database.
	*/
	bool CheckValidUsername(string username);
	/*	Queries from the database if the username is available.
	*/
	bool CheckValidUsernameAndPassword(string username, string password);
	/*	Queries from the database if the username and password is valid.
	*/
	int InsertNewUser (string username, string password, char race);
	/*	Inserts a new entry to the table with the provided arguments. Returns the
		ID of the user provided by the database.
	*/
```

```
void UserEnteredShipToHangar(GameData *data, int user_id, int ship_id);
/*      Finds the user in the database. Finds the ship's old state from the database
        from the ship_id. Finds the ship's new state from the game data by the
        user_id. Updates the data record to match the new state.
*/
void UserBoughtItem(int user_id, int ship_id, char object_id);
/*      Finds the user's ship from the database and adds the specified item to it. Decreases
        the item's amount from the database. Decreases the user's credits according to the
        item's price.
*/
void UserSoldItem(int user_id, int ship_id, char object_id);
/*      Finds the user's ship from the database and removes the specified item from it.
        Increases the item's amount in the database. Increases the user's credits according to
        the item's price.
*/
}
```

class GameData {

```
GameData();
/*      Initializes the game data structure by putting NULL values.
*/
void UpdateObject(Object *object);
/*      Puts the object into the game data by updating the previous record or if there is not a
        previous record inserts the object into the end of the game data.
*/
void SelectFirstObject();
/*      Sets the pointer to the first object in the data.
*/
Object *GetObject();
/*      Gets the pointed object and moves the pointer to the next object. Returns NULL if no
        objects.
*/
void SelectMyFirstObject(int user_id);
/*      Sets the pointer to the first object of the user.
*/
Object *GetMyObject();
/*      Gets the pointed object and moves the pointer to the next object of the user. Returns
        NULL if no objects.
*/
Spaceship *MyShip();
/*      Returns the ship pointed by the game data.
*/
void DeleteDistantObjects(int user_id);
/*      Deletes the objects that are too far from the user's ship.
*/
Object *ReturnMyPerimeter(int user_id);
/*      Allocates new objects for all of the objects in the game data that stay inside the
```

```
        perimeter of the user. Returns the head of the linked list.
    */
}

class MissionModule {
    MissionModule();
    /*  Initializes the mission module.
    */
    void PrepareNewMissions();
    /*  Puts new mission objects to it's list of available missions if it's required.
    */
    void CheckMissionStates(GameData *data, int user_id);
    /*  Checks if the user's accepted mission's state has changed and update related Mission
        object by changing its state if necessary. Delete the mission object if the mission is
        failed or accomplished and the mission object has reached the user.
    */
    void UserAcceptedMission(GameData *data, int user_id, int mission_id);
    /*  Prepare a mission object owned by the user and put it into the game_data.
        Delete the corresponding mission from the list of available missions.
    */
}

class PhysicsModule {
    PhysicsModule();
    /*  Initializes the physics module.
    */
    void MoveMyObjects(GameData *data, int user_id);
    /*  Move all of the user's objects according to their Fx,Fy,Fz,Mx,My,Mz values and
        update their x,y,z,Rx,Ry,Rz values.
    */
    void CheckMyCoordinates(GameData *data, int user_id);
    /*  Check if the coordinates of the user's ship is inside the allowed boundaries of the
        game. Create information instances accordingly.
    */
    void StabilizeMyShip(GameData *data, int user_id);
    /*  Add or subtract from Mx,My,Mz,Fx,Fy,Fz values of the ship in a way creating a
        stabilizer effect.
    */
}

class MultimediaModule {
    MultimediaModule();
    /*  Initializes multimedia module classs.
    */
    void HandleSoundsOfAllObjects(GameData *data, int user_id);
    /*  Checks all of the objects to see if a sound is triggered in that animation frame. If a
        sound is triggered starts playing that sound. If a sound does not exist in the specified
```

```
		frame stops playing the sound. If a sound exists determines the pan and volume of
		the sound by looking at the x,y,z,Rx,Ry,Rz coordinates of the camera and x,y,z
		coordinates of the sound source.
	*/
}
```

class Camera {

```
	Camera(GameData *data);
	/*	Initializes the camera to it's initial place by looking at the coordinates of
		the spaceship.
	*/
	void MoveCameraToItsPlace(GameData *data, int user_id);
	/*	Moves the camera by looking at the current x,y,z values of the camera and by
		deciding where it should be by looking at the x,y,z values of the spaceship of the user.
	*/
}
```

class CollissionDetection {

```
	CollissionDetection();
	/*	Initialize collission detection class.
	*/
	void DetectAllCollissions(GameData *data, int user_id);
	/*	Create an outline of the user's ship and call collission detection function with all of
		the game data that do not belong to the user. If there is a collission, allocate Effect
		objects and put them into the game data.
	*/
}
```

class EffectsProcessorModule {

```
	void HandleEffects(GameData *data, int user_id);
	/*	Finds the effects applied to the user and changes user's health, Mx, My, Mz, Fx, Fy, Fz
		values accordingly. Doesn't delete these handled effects. Then finds the effects that
		the user applied to the other users. Changes user's experience accordingly. Deletes
		these effects from game data.  Creates chat objects that have the user id as the owner
		id if necessary.
	*/
	void HandleExperince(GameData *data, int user_id);
	/*	Checks if the user has leveled up. Creates information instances accordingly.
	*/
	void HandleMyObjects(GameData *data, int user_id);
	/*	Increments frames of all of the user's animated objects. Decrements the life of
		temporary objects like rockets, lasers. If the animation frame requires a callback such
		as the firing of a weapon, the callback is made.
	*/
}
```

```
class InputHandler {
	InputHandler();
	/*	Initializes the input handler class.
	*/
	static void KeyboardUp(char key, int x, int y);
	/*	Called when a key is up.
	*/
	static void KeyboardDown(char key, int x, int y);
	/*	Called when a key is down.
	*/
	static void SpecialKeyboardUp(int key, int x, int y);
	/*	Called when special keys are up.
	*/
	static void SpecialKeyboardDown(int key, int x, int y);
	/*	Called when special keys are down.
	*/
	static void PassiveMouseMove(int x, int y);
	/*	Called when the mouse is moved. The change in mouse coordinates is saved.
	*/
	static void MouseMove(int x, int y);
	/*	Called when the mouse is moved while a mouse button is pressed. The change in
		mouse coordinates is saved.
	*/
	static void MouseClick(int button, int button_state, int x, int y);
	/*	Called when the mouse is clicked. The button that is clicked is saved into a queue.
	*/
	void HandleKeyboardInput(GameData *data, int user_id);
	/*	The effects of the keyboard buttons that are currently down is applied to the ship's
		Fx, Fy, Fz values.
	*/
	void HandleMouseInput(GameData *data, int user_id);
	/*	The change in the mouse coordinates are applied to the ship's Mx, My, Mz values.
		The first click in the mouse click queue is applied if applicable.
	*/
}

class Timer {
	Timer();
	/*	Initializes the timer class and starts the first timer.
	*/
	AddTimerVariable(int *variable, float start_value, float end_value, int time_interval, char
func_type, (void *) function(void));
	/*	Adds a variable to the timer variable list which will be incremented/decremented
		according to the func_type specified. The variable will start with the start_value and
		end with the end_value. If the variable exceeds the end_value the specified function is
		called.
	*/
```

```
        void HandleAllTimerVariables();
        /*      Handles all the declared timer variables and increments/decrements them according
                to their properties.
        */
}

class InterfaceObject {
        InterfaceObject();
        /*      Initializes an interface object.
        */
        abstract void Display();
        /*      Displays the object on the screen.
        */
        abstract bool CheckClick(int x, int y);
        /*      Checks if the mouse clicked within the boundaries of the object.
        */
        void Select();
        /*      Makes the object selected.
        */
        void Deselect();
        /*      Deselects the object.
        */
}

class InterfaceObject :: EditBox {
        EditBox(int x, int y, int width, int height);
        /*      Creates an edit box in the specified place.
        */
        void Display();
        /*      Display the edit box on the screen.
        */
        bool CheckClick(int x, int y);
        /*      Checks if the mouse clicked within the boundaries of the object.
        */
        void EnterChar(char c);
        /*      Enter the char to the edit box.
        */
        void DeleteChar();
        /*      Delete the last char in the edit box. Do nothing if edit box is empty.
        */
}

class InterfaceObject :: Text {
        Text(int x, int y, string text);
        /*      Creates text in the specified place.
        */
        void Display();
```

```
	/*	Display the text on the screen.
	*/
	bool CheckClick(int x, int y);
	/*	Returns false.
	*/
}

class InterfaceObject :: Button {
	Button (int x, int y, int width, int height, string text, (void) function());
	/*	Creates a button in the specified place.
	*/
	void Display();
	/*	Display the button on the screen.
	*/
	bool CheckClick(int x, int y);
	/*	Checks if the mouse clicked within the boundaries of the object.
	*/
	void CallFunction();
	/*	Call the callback function of the button.
	*/
}

class InterfaceObject :: ImageButton {
	Button (int x, int y, int width, int height, BITMAP bitmap_off, BITMAP bitmap_on, (void)
function());
	/*	Creates a button with an image on it in the specified place.
	*/
	void Display();
	/*	Display the button on the screen.
	*/
	bool CheckClick(int x, int y);
	/*	Checks if the mouse clicked within the boundaries of the object.
	*/
	void CallFunction();
	/*	Call the callback function of the button.
	*/
}

class InterfaceObject :: 3DObject {
	3DObject (int x, int y, DISPLAY_LIST 3dobject);
	/*	Creates a 3d object in the specified place.
	*/
	void Display();
	/*	Display the object on the screen.
	*/
	bool CheckClick(int x, int y);
	/*	Returns false.
```

```
	*/
	void Rotate();
	/*	Rotate the 3d object 1 degree around z axis.
	*/
}
```

class UserInterface {

```
	UserInterface(BITMAP background);
	/*	Create a user interface with the specified background.
	*/
	void Activate();
	/*	Registers the display callback of the program to the display function and the
		keyboard and mouse callbacks to the corresponding functions.
	*/
	void Display();
	/*	Draw the background and all of the interface objects added to the user interface.
	*/
	void AddInterfaceObject(InterfaceObject *object);
	/*	Add the object to the user interface.
	*/
	static void KeyboardUp(char key, int x, int y);
	/*	Called when a key is up.
	*/
	static void KeyboardDown(char key, int x, int y);
	/*	Called when a key is down.
	*/
	static void SpecialKeyboardUp(int key, int x, int y);
	/*	Called when special keys are up.
	*/
	static void SpecialKeyboardDown(int key, int x, int y);
	/*	Called when special keys are down.
	*/
	static void PassiveMouseMove(int x, int y);
	/*	Called when the mouse is moved. The change in mouse coordinates is saved.
	*/
	static void MouseMove(int x, int y);
	/*	Called when the mouse is moved while a mouse button is pressed. The change in
		mouse coordinates is saved.
	*/
	static void MouseClick(int button, int button_state, int x, int y);
	/*	Called when the mouse is clicked. The button that is clicked is saved into a queue.
	*/
}
```

class Object :: Effect {

```
	Effect(int owner_id, int effected_id, char effect_type);
	/*	Initializes the effect class.
```

```
	*/
	void Display();
	/*	Does nothing.
	*/
	char GetEffectType();
	/*	Returns the effect type.
	*/
	int GetEffectOwner();
	/*	Returns the effect owner.
	*/
	int GetEffectedUser();
	/*	Returns the effected user.
	*/
}
```

class Object :: Spaceship {

Spaceship(int owner_id, int ship_id, char ship_type, int owner_experience, float x, float y, float z, float Rx, float Ry, float Rz, float health, float shield, int body_level, char body_state, char body_frame, int laser_level, char laser_state, char laser_frame, int rocket_level, char rocket_state, char rocket_frame, int turret_level, char turret_state, char turret_frame, int mine_level, char mine_state, char mine_frame, int magnet_level, char magnet_state, char magnet_frame, int stealth_level, char stealth_state, char stealth_frame, char selected_weapon, int rocket_amount, float rocket_health, int mine_amount, float mine_health, float laser_health, float turret_health, int magnet_amount, float magnet_health, int stealth_amount, float stealth_health);

```
	/*	Initializes the spaceship with the given values.
	*/
	void Display();
	/*	Displays the ship and it's weapons.
	*/
	Vertex GetCoordinates();
	/*	Gets the x, y and z coordinates of the ship.
	*/
	void SetCoordinates(Vertex data);
	/*	Sets the x, y and z coordinates of the ship.
	*/
	Vertex GetForce();
	/*	Gets the x, y and z force vectors of the ship.
	*/
	void SetForce(Vertex data);
	/*	Sets the x, y and z force vectors of the ship.
	*/
	Vertex GetRotation();
	/*	Gets the x, y and z rotation values of the ship.
	*/
	void SetRotation(Vertex data);
	/*	Sets the x, y and z rotation values of the ship.
	*/
```

```
	Vertex GetMoment();
	/*	Gets the x, y and z moment vectors of the ship.
	*/
	void SetMoment(Vertex data);
	/*	Sets the x, y and z moment vectors of the ship.
	*/
	char GetSelectedWeapon();
	/*	Returns the selected weapon.
	*/
	void SetSelectedWeapon(char weapon_type);
	/*	Sets the selected weapon.
	*/
	void ReduceSelectedWeapon();
	/*	Reduces 1 from the amount of the selected weapon.
	*/
	void CanFireSelectedWeapon();
	/*	Returns true if the selected weapon's amount is larger than 0.
	*/
	int GetShipID();
	/*	Returns the ship's id.
	*/
}
```

class Object :: Rocket {

```
	Rocket(int owner_id, int rocket_id, int rocket_life, char rocket_level, char rocket_state, char rocket_frame, float x, float y, float z, float Rx, float Ry, float Rz);
	/*	Initializes the rocket class.
	*/
	void Display();
	/*	Displays the rocket.
	*/
	bool CheckCollissionWithShip(Object *ship, Vertex *collissionPoint);
	/*	Detects the collission between the ship and the rocket and writes the collission point
		to the vertex. Returns true if there is a collission, false if there is not. To optimize the
		speed of collission detection the length between two objects is first checked.
	*/
}
```

class Object :: Laser {

```
	Laser (int owner_id, int laser_id, int laser_life, char laser_state, char laser_level, char laser_frame, float x, float y, float z, float Rx, float Ry, float Rz);
	/*	Initializes the laser class.
	*/
	void Display();
	/*	Displays the laser.
	*/
	bool CheckCollissionWithShip(Object *ship, Vertex *collissionPoint);
```

/*	Detects the collission between the ship and the laser and writes the collission point to the vertex. Returns true if there is a collission, false if there is not. To optimize the speed of collission detection the length between two objects is first checked.
*/
}

class Object :: Mine {
Mine (int owner_id, int mine_id, int mine_life, char mine_state, char mine_level, char mine_frame, float x, float y, float z, float Rx, float Ry, float Rz);
/*	Initializes the mine class.
*/
void Display();
/*	Displays the mine.
*/
bool CheckCollissionWithShip(Object *ship, Vertex *collissionPoint);
/*	Detects the collission between the ship and the mine and writes the collission point to the vertex. Returns true if there is a collission, false if there is not. To optimize the speed of collission detection the length between two objects is first checked.
*/
}

class Object :: Mission {
Mission(int mission_id, char mission_type);
/*	Initializes the mission. Called by the mission module when it is first created.
*/
Mission(int owner_id, int mission_id, char mission_type, char mission_state);
/*	Initializes the mission. Called by the decode module when it is first created.
*/
void Display();
/*	Does nothing.
*/
char GetMissionState();
/*	Returns the mission state.
*/
char SetMissionState();
/*	Sets the mission state.
*/
char GetMissionType();
/*	Returns the mission type.
*/
}

class Object :: Chat {
Chat(string chat);
/*	Initializes the chat.
*/
void Display();

```
    /*      Does nothing.
    */
    string *GetChat();
    /*      Returns the chat string.
    */
}
```

class ChatBox {

```
    ChatBox(GameData *data, int x, int y);
    /*      Creates a chat box in the specified coordinates.
    */
    void Display();
    /*      Finds the chat object in the game data that has the owner id 0 and appends it to the
            chat box buffer. Deletes the chat object. If send boolean is true the input buffer of the
            chat box is created as a new chat object. Then the buffer is emptied. Then it displays
            the chat box.
    */
    void SendChat();
    /*      Sets a send boolean true to send the buffer on the first ProcessChat call.
    */
    void EnterChar(char c);
    /*      Appends a char to the input buffer.
    */
    void DeleteChar();
    /*      Deletes a char from the end of the input buffer. Nothing is done if the buffer is empty.
    */
}
```

class MapBox {

```
    MapBox(GameData *data, int user_id, int x, int y);
    /*      Creates a map box in the specified coordinates.
    */
    void Display();
    /*      Displays the map box by displaying all the displayable objects as dots with
            corresponding colors.
    */
}
```

class HealthBox {

```
    HealthBox(GameData *data, int user_id, int x, int y);
    /*      Creates a health box in the specified coordinates.
    */
    void Display();
    /*      Displays the health of the user with the health box.
    */
}
```

class ShieldBox {
ShieldBox(GameData *data, int user_id, int x, int y);
/* Creates a shield box in the specified coordinates.
*/
void Display();
/* Displays the shield of the user with the shield box.
*/
}

class FuelBox {
FuelBox(GameData *data, int user_id, int x, int y);
/* Creates a fuel box in the specified coordinates.
*/
void Display();
/* Displays the fuel of the user with the fuel box.
*/
}

class SpeedBox {
SpeedBox(GameData *data, int user_id, int x, int y);
/* Creates a speed box in the specified coordinates.
*/
void Display();
/* Displays the speed of the user with the speed box.
*/
}

class SelectedWeaponBox {
SelectedWeaponBox(GameData *data, int user_id, int x, int y);
/* Creates a selected weapon box in the specified coordinates.
*/
void Display();
/* Displays the selected weapon of the user with the selected weapon box.
*/
}

class InformationBox {
InformationBox(int x, int y);
/* Creates an information box in the specified coordinates.
*/
void PutInformation(string info, int time, int priority);
/* Replaces the previous information if this information's priority is of higher value. If no information exists the string is saved. If the same information exists updates it's time value.
*/
void Display();
/* Display the current information if it exists and decrease it's time value. If the time

```
		value is beyond 0 delete the information.
	*/
}

class ExperienceBox {
	ExperienceBox(GameData *data, int user_id, int x, int y);
	/*	Creates an experience box in the specified coordinates.
	*/
	void Display();
	/*	Display the experience box.
	*/
}

class EnemyDetails {
	EnemyDetails(GameData *data, int user_id);
	/*	Initializes enemy details class.
	*/
	int FindSelectedShip();
	/*	Apply 3d picking from center of the spaceship to the front of the spaceship. If there is
		a ship it returns the ship's user id. This user id is also saved into a place. If there is
		already the same user id saved it increments a counter which represents the number
		of times this ship was spotted in front of the ship.
	*/
	bool Locked();
	/*	Returns true if the saved counter is more than a predetermined number.
	Puts an information if locked.
	*/
	void Display();
	/*	If there is a selected ship displays it's properties around it.
	*/
}

class ClientLoop {
	ClientLoop();
	/*	Initializes the class.
	*/
	void MainLoop();
	/*	Coordinates the calling of the client loop functions.
	*/
	int DirtyPixelLogoLoop();
	/*	Coordinates the displaying of the dirty pixel logo. Returns a message to the main
		loop representing the ending state of the loop.
	*/
	int TwilightLogoLoop();
	/*	Coordinates the displaying of the twilight logo. Returns a message to the main loop
		representing the ending state of the loop.
	*/
```

```
	int LoginScreenLoop();
	/*	Coordinates the displaying of the login screen. Returns a message to the main loop
		representing the ending state of the loop.
	*/
	int RegisterScreenLoop();
	/*	Coordinates the displaying of the register screen. Returns a message to the main loop
		representing the ending state of the loop.
	*/
	int LoadingScreenLoop();
	/*	Coordinates the displaying of the loading screen. Returns a message to the main loop
		representing the ending state of the loop.
	*/
	int RaceSelectionScreenLoop();
	/*	Coordinates the displaying of the race selection scren. Returns a message to the main
		loop representing the ending state of the loop.
	*/
	int HangarScreenLoop();
	/*	Coordinates the displaying of the hangar screen. Returns a message to the main loop
		representing the ending state of the loop.
	*/
	int BuyScreenLoop();
	/*	Coordinates the displaying of the buy screen. Returns a message to the main loop
		representing the ending state of the loop.
	*/
	int SellScreenLoop();
	/*	Coordinates the displaying of the sell screen. Returns a message to the main loop
		representing the ending state of the loop.
	*/
	int MissionSelectionScreenLoop();
	/*	Coordinates the displaying of the mission selection screen. Returns a message to the
		main loop representing the ending state of the loop.
	*/
	int GameplayLoop();
	/*	Coordinates the displaying of the game screen. Returns a message to the main loop
		representing the ending state of the loop.
	*/
}

class ServerLoop {
	ServerLoop();
	/*	Initializes the server loop class.
	*/
	void MainLoop();
	/*	Coordinates the server loop.
	*/
}
```

```
class AIModule {
	AIModule();
	/*	Initializes the AI module class.
	*/
	void CreateNewNpcs(GameData *data);
	/*	Creates new npc characters if NPC count / PC count ratio is below some  limit.
		Assigns npc ids for each NPC.
	*/
	void HandleNpcEffects(GameData *data);
	/*	Handles the effects that are related to npcs.
	*/
	void DetermineNpcsState(GameData *data);
	/*	Determines the AI state for each NPC in the game data.
	*/
	void HandleNpcs(GameData *data);
	/*	Make the npcs move and shoot according to their states.
	*/
}
```

8. Testing Issues

i. Test Design

Since we haven't gained full insight of what must be done in test cases in all o our modules we only designed a test case for the network modules which are nearly complete. As we progress we'll design different test cases for the other modules of our implementation.

ii. Test Cases

A scenario for the packet size:

- Lets assume there are "n" players in the game visible area of the user.
- And for the worst case lets assume that in every time increment all of the objects in the universe are changing. A player will get the information of "n-1" other ships for this case which is 128 bytes(14 int, 14 float, 16 char) and also information of his/her ship which is also 128. The total value for the whole information is 128 x n.
- Lets assume every player fired at most one weapon in the previous time increment (this is an optimistic assumption). In this case there will be "n" ammos flying in the universe. That is n x 39 bytes(3 int, 3 char, 6 float) for any kind of weapon ammo.
- Lets also assume that every player received damage from a previous ammo released by a player. For this case there must be 39 x n bytes for the ammos, since their animations are not completed yet. There must be also "n" effect objects in the packet which counts 9 x n bytes(2 int, 1 char) more. Then a total of 48 x n bytes for this case.
- There is also 1 byte for the packet header.

Then assuming there are no NPC's in the visible area 215 x n bytes must be sent to a single user.

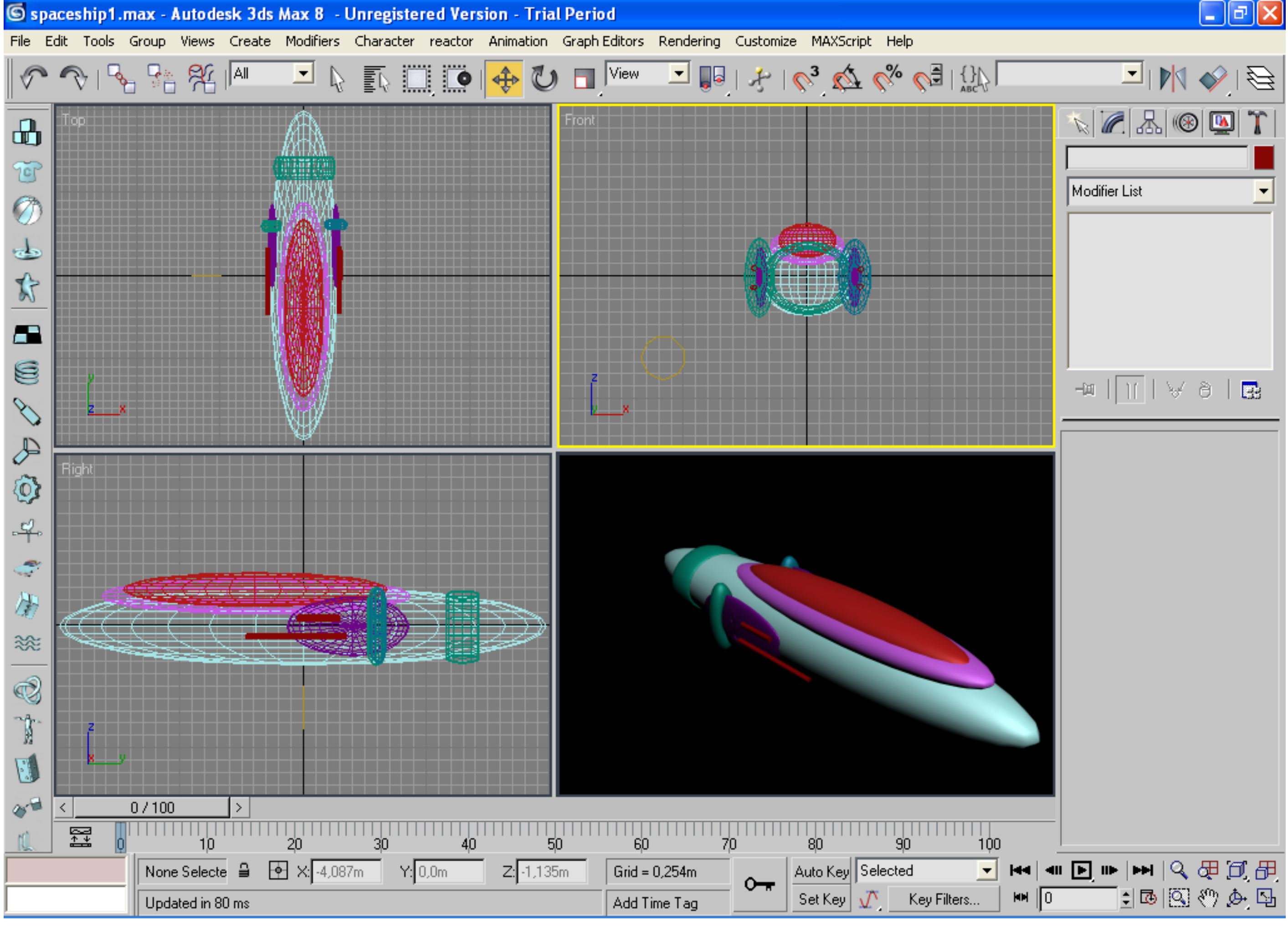
Assuming there are 10 players in the visible area the packet size is 2150 bytes ≅ 2.1 kbytes
Then for a frame rate of 20 fps 2.1 x 20 = 42 kb's must be sent to the player in one second assuming the scenario is always the same in that second(every player is firing a weapon and gots hit at the same time).

Since the server will only send information about the objects which are visible to the user, (which is another optimization to speed up the messaging) this calculations are realistic. For a war scene of 10 players and a scenario like this at least 0.5 mbits of an internet connection is needed.

This scenario can be tested by creating virtual packets (a dummy packet) of 2.1 kb's and sending them to 10 clients at the same time.

9. Appendix

i. Model Spaceships



spaceship2.max - Autodesk 3ds Max 8 - Unregistered Version - Trial Period

File Edit Tools Group Views Create Modifiers Character reactor Animation Graph Editors Rendering Customize MAXScript Help

All View

Perspective

Perspective

Left

Modifier List

0 / 100

10 20 30 40 50 60 70 80 90 100

None Selecte X: 0,0m Y: 1,231m Z: -0,709m Grid = 0,254m Auto Key Selected

Click and drag up-and-down to zoom in and out Add Time Tag Set Key Key Filters... 0

spaceship32.max - Autodesk 3ds Max 8 - Stand-alone License

File Edit Tools Group Views Create Modifiers Character reactor Animation Graph Editors Rendering Customize MAXScript Help

All View

Front

Perspective

Top

Standard Primitives

- Object Type

AutoGrid

Box Cone

Sphere GeoSphere

Cylinder Tube

Torus Pyramid

Teapot Plane

- Name and Color

0 / 100

10 20 30 40 50 60 70 80 90 100

None Selecte X: Y: Z: Grid = 10,0 Auto Key Selected

Click and drag up-and-down to zoom in and out Add Time Tag Set Key Key Filters... 0

Task Name	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14	Week 15	Week 16	Week 17	Week 18	Week 19	Week 20	Week 21	Week 22	Week 23	Week 24
Game Concept Development																								
* Storyline																								
* Gameplay design									¬															
* Interfaces design																								
* Sketching									¬															
* Objects design																								
* Objects modelling																								
Graphics Engine Development																								
* Engine basics creation							¬																	
* Displaying 3ds objects																								
* Displaying animations																								
* Displaying lights																								
Network Modules Development																								
* Chat module																								
* Packet creation																								
* Packet parsing																								
* Packet transfer						¬																		
* Send and Receive modules																	¬							
* Synchronization																								
Data Structures Development																								
* Organization of data									¬															
* Encapsulation of data																								
* Database tables creation																								
* Database interaction classes																								
Artificial Intelligence Development																								
* NPC design																								
* NPC behavior modelling															¬									
* NPC behavior programming																								
Game Flow Development																								

* Maps design																								
* Maps programming																								
* Quests design																								
* Quests programming																								
* Game flow programming																								
Deployment and Testing																								
* Deployment of classes																								
* Designing test cases																								
* Testing																								