COP 290 - Assignment 3
Battlestar Galactica

Akshay Kumar Gupta
2013CS50275

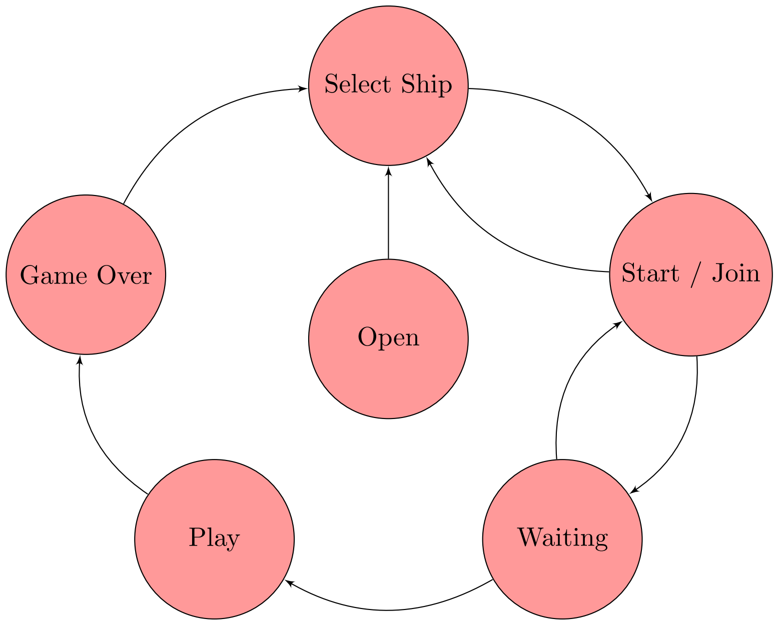
Haroun Habeeb
2013CS10225

Barun Patra
2013CS10773

J. Shikhar Murty
2013EE10462

Abstract

The game is a real-time racing cum FPS game. It takes place in space. Each player owns a spaceship and the objective is to be the first to reach the end. Players can also shoot at other players and try and destroy their ships. Each spaceship has different stats like accuracy, speed, ammo etc. The game is designed for a maximum of four human and four AI players.



Game Loop

1 Overall Design

2 Network

We will use UDP to handle network connections. It is a completely connected peer-to-peer network.

2.1 Start of game

One user starts the game and the P2P network. This user specifies how many people will be joining the game. A new user can connect to the network by sending his IP to any existing player in the network. As UDP is being used there is no notion of an actual ‘connection’, all each user needs is the IP address of each other player. The existing player will give the IP of the new user to all other users and the IP of all other users to the new user.

2.2 Message Passing

- A user broadcasts his state to all other users at regular time intervals.
- We have a separate thread for listening to incoming messages.
- When a new message is received from another user it is passed on to the rendering thread which renders that user according to his state.
- In case another user sends us a message which drops, the new position of the user is simply extrapolated from his old position, velocity, and acceleration.

2.3 Handling Drops

If we have not received a message from a user for the last T seconds, then we assume that the user has dropped. T is large enough compared to the average time interval between messages for the probability to be very low that the user has not dropped.

Listing 1: Network Manager

```
class NetworkManager {

public:
   void init(PlayerInfo& pinfo); //Create player
   void nullify();
   bool createNetwork(); //Creates a port and starts listening.
   bool connectToNetwork(tIP4 peerIP); //Create a port, listen on
       it and also send message to peerIP etc.
   bool leaveNetwork(); //Close the active port.
   bool isAlive(int p_idx);
   bool killIdlers(); //Also broadcasts the message across.
```

```
    bool broadcastMessage(); //send message to all.
    bool sendMessage();
    bool getMessages(std::vector< GameEvent >& eventQueue); //All
        the events that need to be processed for the next rendering
        iteration.
    /* Debugging functions: The boolean is to toggle debug mode */
    void dprint(bool debug_net=false);
    void drender(bool debug_net=false);

private:
    //vector of IPs of all players.
    PlayerInfo me;
    int myIdx;
    std::vector<PlayerInfo> network;
};
```

3 Graphics

3.1 Model Rendering

- Blender will be used to create models of spaceships, asteroids, debris etc.
- We will load the object files for these models using our own code.
- We will also have our display function for each model.
- However, the bodies actually have bounding boxes. These bounding boxes are what the physics engine will handle.

Listing 2: Space Object

```
class SpaceObject {
public:
    SpaceObject() {}
    void init(OBJECT_TYPE _objtype); //Creation Loading
    //Getters/Setters
    Position getPosition();
    Orientation getOrientation();
    Velocity getVelocity();
    Acceleration getAcceleration();
    void setPosition(Position& p);
    void setOrientaiton(Orientation& _o);
    void setVelocity(Velocity& _v);
    void setAcceleration(Acceleration& _acc);
    //Rendering functions
    void loadOBJ();
    void renderOBJ();
    //Queries
    bool inside(Position& p);
    bool outside(Position& p);
    bool onsurface(Position& p);
    bool intersects(LineSeg& l);
    bool intersects(Ray& r);
    bool intersects(Line& l);
    bool intersects(Cuboid& cub);
    bool intersects(SpaceObject& spaceobj);
    //Movement
    glm::vec3 translate(glm::vec3 displacement); //Translate object
    glm::vec3 rotate(glm::vec3 rotation); //Rotate object
    glm::vec3 scale(glm::vec3 vec_scale); //Scale object
    // Debugging functions: The boolean is to toggle debug mode.
    void dprint(bool debug_spaceobj=false);
    void drender(bool debug_spaceobj=false);
private:
    std::vector<Cuboid> pieces;
    OBJFilepath objFile;
    OBJECT_TYPE objType;
    //Properties
    Mass mass;
    Position com;
    Orientation o;
    Velocity v;
    Acceleration a;
};
```

3.2 User Interface / HUD

We will offer a certain degree of customizability of the UI to the user.

- Select Ship : On this screen, the user sees what ships he can select, we may modify some properties of the ship - such as the droid. he can alter the mouse sensitivity and the keyboard mappings. There'll also be a *Play* button that takes the user to the Start/Join screen.
- Start/Join : This screen allows the user to either
  1. Start: He can start the game with some options, such as the number of players and the number of AI.
  2. Join : He can join a game by providing the IP of any member of a network.
- Waiting : This is the screen shown before the match actually begins. On this screen, the user waits while more people join the match.
- Play : This is what is displayed while the player actually plays the game. The world is rendered with a HUD. The HUD displays properties such as health, ammo.

All of these screens will be implemented using openGL / freeGLUT / GLUI / myGUI.

3.3 Camera Positions

According to the ship, various camera positions will be available. For example, the TIE fighter might have only two views - first person and the third person. The X-Wing will have an additional Droid-view. The user can toggle between these views during the game. This can be handled by simply moving the camera of that user.

4 Music

Music will be incorporated in our project using openAL. Every client will have a seperate thread to play the music.

5 Physics and Collision Detection

5.1 Collision Detection and Handling

- Players' ships can collide with other ships and floating space debris. Our code needs to be able to detect and handle such collisions. We will use bounding boxes for detection of collisions.
- Once a collision has been detected, using the point of contact and vectors along collision, we can resolve the equations and assign the bodies their respective velocities.

5.2 Hit Detection

- When a user shoots his weapons, we need to check if the projectile hit any of the players. This again, will be done by using bounding boxes for each ship.
- If a hit has happened, both parties involved will get a visual cue.

Listing 3: Physics

```cpp
class LineSeg {
public:
   Position start, end;
   Length length;
   LineSeg();
};

class Ray {
public:
   Position start;
   Direction direction;
   Ray();
};

class Line {
public:
   Position start; //Every line must pass through a point.
   Direction direction; //Extends in both directions.
   Line();
};

class Cuboid { //acts as the bounding box for everything.
public:
   Cuboid() {}
   glm::vec3 translate(glm::vec3 displacement); //Translate cuboid
   glm::vec3 rotate(glm::vec3 rotation); //Rotate cuboid
   glm::vec3 scale(glm::vec3 vec_scale);  //Scale cuboid
   // Queries to the cuboid
   bool inside(Position& p);
   bool outside(Position& p);
   bool onsurface(Position& p);
   bool intersects(LineSeg& l);
   bool intersects(Ray& r);
   bool intersects(Line& l);
   bool intersects(Cuboid& cub);
   //Collision handlers
   std::pair<Position,Direction> solveCollision(LineSeg& l);
   std::pair<Position,Direction> solveCollision(Ray& r);
   std::pair<Position,Direction> solveCollision(Line& l);
   std::pair<Position,Direction> solveCollision(Cuboid& cub);
   // Debugging functions: The boolean is to toggle debug mode.
   void dprint(bool debug_cuboid=false);
   void drender(bool debug_cuboid=false);
   // Getters and setters.
   Position getCentre();
   Dimension getDimension();
   Orientation getOrientation();
   void setCentre(Position& _p);
   void setDimension();
   void setOrientation();
private:
   Position centre;
   Dimension dimensions;
   Orientation orientation;
};
```

6 Players & AI

6.1 Player

Each player has a unique Player No. The different properties of the player are :

- Player settings : This includes the player's keyboard and mouse preferences, colour choices etc.
- Player spaceship : The spaceship that the player chooses.
- Player state : This is a tuple of

Listing 4: Player

```
class UserSettings {
public:
   glm::vec3 mouseSensitivity;      //Along x,y,z.
   KeyboardMapping keyboardMapping;
   UserSettings() {}
   void read_settings(); //Read from setting files.
   void save_settings(); //Save to setting files
};

class Player {
public:
   Player() {}
   // Debugging functions: The boolean is to toggle debug mode.
   void dprint(bool debug_cuboid=false);
   void drender(bool debug_cuboid=false);
   // glut-linking-event handlers.
   void keyboardHandler(); //Takes in keyboard input.
   void mouseHandler(); //Takes in mouse position.
   void shootHandle(); //Takes in clicked point.
   void shotHandle(GameEvent); //Handle getting shot.
   void updateGameState();

private:
   // Loaded upon creaiton.
   UserSettings settings;
   NetworkManager manager;
   std::vector< GameEvent > todolist; //List of all messages
       received etc.
   WorldState world;
};
```

6.2 AI

The AI is another player in the race. It is modelled as a utlity based agent handling partial observability.

- Each user will host at most 1 AI.

- The game state space can be formulated as a *Markov Decision Process*.
- The AI will use reinforcement learning to choose the best action at every point of time.
- The reward function is fairly simple - a high reward for winning, a good reward for scoring a hit, a neutral reward for existence and a punishment for dying.
- The difficulty level is essentially how long the AI Player has learnt. There will be three difficulty levels - *easy*, *medium* and *hard*.

7 Testing

7.1 Network

- We will test that message passing on the UDP network works correctly.
- We will test whether the computers correctly identify that a certain player has dropped.

7.2 Graphics

- We will visually check that models are loaded into openGL and rendered correctly.
- We will check that all mouse and keyboard controls work correctly.
- We will test that different camera views from the spaceship are accurate.

7.3 Physics

- We will carry out extensive unit testing to check that collision detection and handling works correctly.
- We will carry out unit testing to check that calculation of new position and velocity based on a player state is correct.

7.4 AI

- We will play against our AI to test how good it is.
- We will check whether the three levels correctly reflect how the AI's actually play.

8 Planned Improvements