CS 447/547: Game Design Project 2

Readme

Travis Hall
travis.hall@email.wsu.edu

Bhadresh Patel
bhadresh.patel@email.wsu.edu

Michael Persons
michael.persons@email.wsu.edu

Washington State University Vancouver
Fall, 2009

Abstract

Iron Legends is a top-down arcade-style tank combat game. The goal is simple: total domination! Players will strap themselves into the cockpit of a tank and blast, smash, and detonate anyone and everyone who dares to stand in their way. With two different modes of gameplay–single-player and networked multiplayer–combatants can pit their wits and their bravery against legions of computer-controlled tanks or test their mettle against friend and foe alike. Let the mayhem begin!

1 Project Implementation Status

1.1 Low Bar

- Two maps
- Player Base
- Mini map/Radar
- Obstacles: Wall, Trees, Rocks
- Power-ups: Shield, Double cannon, Speed, Armor, Extra Life
- AI Tanks for single player mode
- Single/Multiplayer integration
- Multiplayer: Server create, lobby, packet send/receive architecture.
- Sound effects and music

1.2 High Bar

- Movable turret
- Collectible powerups

1.3 Additional Features implemented

- Map Editor

2 Technical Showpieces

2.1 Multiplayer via Networking

Unfortunately, this technical showpiece is only mostly-complete. As we went to hook the networking into the multiplayer, we discovered extremely subtle problems that cropped up with the Java NIO classes (i.e. SocketChannel, Selectors, etc.) and the network. What worked in one environment failed to work on another.

After experiencing these major setbacks, we spent a good 10-12 hours on Sunday completely reworking the networking to use Multicast messages. While this works (and can be demonstrated

on the Multiplayer Server Select and Lobby screen), the packets aren't handled once into the multiplayer game situation and so game updates aren't accepted and used. This is only an hour or two from being completed, but alas, we ran short of time.

As an aside, because it's so frustratingly close to completion, it may be completed sometime in the future (in fact, as of writing this, there is a push containing movement networking, waiting for GitHub to finish a repository migration). Keep an eye on the GitHub repository! http://github.com/doggles/Iron-Legends

2.1.1 NIO Details

Nitty gritty implementation details can be seen in jig/ironLegends/oxide/client and [...]/oxide/server under the ILNIO*.java classes.

With the client, the ROX NIO tutorial (http://rox-xmlrpc.sourceforge.net/niotut/) was followed quite heavily. In fact, the queueing system and the change request system was followed to the letter. Each message was broken up into constituent packets (of 1400 byte max-length) that could be created and reassembled by way of a packet factory. Any packets that were split could also be reassembled, so packet size was not a concern (though they never seemed to get above about 900 bytes even with a full set of data).

The server also began from the ROX NIO tutorial, but ended up diverging heavily, doing away with the change request system. The server simply registered an interest operation on reading or accepting connections (though only one at a time). When the tick expired, the server would then look at the list of connected clients (created when it accepted a connection), and update them directly through their open channels.

In order to discover the available servers, the client and server were given an ILAdvertisementSocket that handled reading and writing to a multicast port (this became the main data socket after the rebuild, since it was convenient and already created). By reading on the multicast group, the client reassembled an ILAdvertisementPacket and could use that for determining the host's address, server name, map name, number of players, and other information.

When the "connect" button is hit, the client would request a connection on the server's channel and finalize the connection—though there was one nuance where a client connecting could potentially finalize the connection immediately (if it were on localhost for example), so there was a requirement to handle that. The Server would then create a ClientInfo for this player, attach the ClientInfo to the key (so the key could be used as an identifier – they're not hashable it seemed so attaching was the only other course of action) as well as put it in a list (for later mass-updates), and then begin sending lobby updates.

This is where the trouble started to come up. In initial testing, this worked perfectly. Then as we were attempting to use it on the campus network, packets stopped being received despite being sent (i.e. the server didn't receive client updates and the client didn't receive server updates). On occasion, the packets would be received, but at a tremendously slow rate. As a group, we spent a few hours quashing potential bugs that we noticed in the code, but ultimately nothing was fixed. This was when we decided that we needed to try another approach.

We completely rewrote the server and client using the multicast sockets and the existing ILAdvertisementSocket class. This fixed a few things, but ultimately, the clients and the servers were still getting slow updates. After a bit of trying to troublshoot this, we decided to split the ILAdvertisementSocket across two ports — one for reading, one for writing. This seemed to fix latency issues, the servers could host and the clients could connect, but at that point it was rather late, the

campus was closing, and so we called it a day before completing the implementation of the game update packets. Unfortunately, we weren't quite fast enough the next day to get the networking fully completed (though it was very close).

2.2 Artificial Intelligence

In Singleplayer mode player play's with the computer controlled tank. The computer controlled tanks are navigated by using Artificial Intelligence (AI) techniques. To make the tank movements more realistic we choose to use behavior based AI.

Following behaviors has been implemented in the game:

Wander Wander is a type of random steering. To make sure the movement is not "twitchy" it retain steering direction state and make small random displacements to it each frame. To achieve this, we constrain the steering force to the surface of a sphere located slightly ahead of the tank. To produce the steering force for the next frame a random displacement is added to the previous value, and the sum is constrained again to the sphere's surface. Tanks starts with this behavior and wander around in the world. If tank detect the opponent tank in its radar then it switches to Arrive behavior.

Seek Seek acts to steer the tank towards a target. This behavior adjusts the character so that its velocity is radially aligned towards the target.

Arrive Arrive behavior is identical to seek while the character is far from its target. But instead of moving through the target at full speed, this behavior causes the character to slow down as it approaches the target, eventually slowing to a stop.

Flee Flee is simply the inverse of seek and acts to steer the character so that its velocity is radially aligned away from the target. The desired velocity points in the opposite direction.

Obstacle Avoidance Obstacle avoidance behavior gives a tank the ability to maneuver in a map by dodging around obstacles. It predicts the future position based on the current velocity and look for possible collision, if it can hit an obstacle then try to move away from the obstacle. To make it more realistic the change in velocity are done gradually. It also checks if it the tank is stuck for a while, then choose new random target and try to move away from.

2.3 Dual Mode: Single/Multi Player

As mentioned earlier, the multiplayer connection was moments from completion. However, there was a nice divide already set up such that the individual who chose to host a server, effectively just duplicated the actions of the server. The single player game would simply run a server and then run the AI in addition. The single/multiplayer win/lose scenarios already existed and could be swapped between nicely as well.

3 Cheat Codes

There are following cheat codes left intentional in the game for reviewer. Press the key described in Key column below to achieve the effect.

Key	Description
0	God Mode - Player tank is indestructible
1	Execute power - shield
2	Execute power - upgrade
3	Execute power - double cannon
9	Kill your tank

Table 1: *Iron Legends*: Cheat Codes

4 License

Iron Legends source code is freely distributable under the terms of the MIT license[1]. Please see included LICENSE file for terms of use for the game and additional attribution.

[1] http://www.opensource.org/licenses/mit-license.php