AS-0.1102 -- Project Plan

Our very own -- Yet Another Liero Clone

Authors

(Almost) Full Name	Student ID	Email
Tero Juhani Marttila	78949E	tjmartti@cc.hut.fi
Atle Juhani Kivelä	79171V	atle.kivela@tkk.fi
Marko Olavi Rasa	78726L	morasa@cc.hut.fi
Eric Emil Malmi	80351A	emalmi@cc.hut.fi

Last updated

2008-11-05

Requirement specification

The aim of our project is to implement a clone of Liero with additional features (e.g. network multiplayer). In general terms, a real-time 2D shooter with multiple players and various weapons, network-based multiplayer and realistic physics.

Core Requirements

- Runs on a Linux platform
- Real time gameplay
- 2D engine/graphics
- Network multiplayer
- Keyboard movement and aim
- Multiple weapons
- Deathmatch mode
- Jumping and ninja rope
- Destructible terrain (weapons, digging)
- Basic artificial intelligence
- Simple map generator

Optional Requirements

- Runs on a windows platform
- Single-host multiplayer (two players on the same screen/keyboard)
- Alternative mouse-based aiming
- Other game modes (capture-the-flag, team deathmatch, races)
- Sound effects/audio
- Multiplayer chat
- Load content from server (maps, scripts, sound effects, etc)
- Advanced physics (flexbile rope, fluid water, collapsing terrain, wind)
- Scripting (for custom weapons, player models, AI, map generator, missions with rules)
- Map editor
- Advanced terrain/materials (layering, flammable terrain, rocks rolling downhill)
- Limited visibility (line-of-sight, darkness, light sources)
- Vehicles (VROOM VROOM)

Use of External Libraries

- SDL?
- ClanLib?
- Boost?

Program architecture

The game is split into several distinct components, designed to be able to separate the server/client multiplayer behaviour from the client-side graphics/user-input functionality. The architecture should enable the development and testing of the various components to be relatively independent of each other.

The network multiplayer will be based on a client-server architecture, whereby clients can make predictions/guesses about future states, but there is a single server which can make authorative decisions about the order and timing of events.

At the core of the program architecture sits the GameState component. It has various APIs and modes of operation, so that it can function in varying multiplayer roles:

- As a network server with or without local players (human or AI)
- As a network client with one (or possibly several) local human/AI players, interacting with other players via a remote network server
- As a stand-alone client with several local human/AI players, not communicating with any remote network server.

The GameState component interfaces with the graphics, user-input, AI, physics and network components. It stores all the information that needs to be replicated between different clients and the server

- For stand-alone clients, the implementation can be fairly simple as the state of all players is know locally.
- For network servers, it can know a single authorative current state, but may also need to track past states for dealing with clients.
- For network clients, it can store the version of the game state the server has last transmitted to it, but it may also need to predict future states to keep the local UI responsive.

In all cases, the behaviour of the graphics and AI components is the same. The GameState can provide a view of the current state, and the graphics/AI components do not need to know if this is a client-side prediction or an authorative stand-alone version (other than the fact that the state may change in strange ways as guesses are fixed).

The user-input/AI actions are simply handed to the GameState, which can then handle it correctly based on our current role. For stand-alone clients or players running on the server, the actions can be applied directly, but for network clients, the GameState can use this input to form predictions about future states, but must send the action to the server, which can then validate it correctly. In all cases, the network server must then send this updated GameState to all network clients.

The physics component is more tightly integrated with the GameState internals, as it has various tasks to do. It must be able to calculate the current to-be-rendered state based on time-series data from known states and predicted states, validate and calculate the effects of various actions, and calculate events like collisions.

The network component is then responsible for replicating the GameState across the network clients and server. It must be able to transmit actions from the clients to the server, and then transmit updates to the game state from the server to the clients. It must be able to handle and recover from things like latency and packet loss without corrupting the game states completely, but due to the real-time aspect, the client's GameStates will probably never be 100% accurate, but rather good approximations. Additionally, it must be able to handle the management of

network games (create server, join server, new client, client gone, etc) and non-real-time actions like player chat, downloading content, etc.

This architecture enables the development and testing of the various components to be relatively independent of each other, once the fundamental GameState component has been built.

Core Components

Physics

Interpolate/extrapolate between time-series states to provide more fluid movement for players/weapons, using e.g. Numerical Runge Kutta integration. ClanLib offers ready-made vector-based collision detection, which would require the game terrain/objects to be represented using line segments. However, implementing some kind of classical pixel-based terrain and collision detection ourself would probably be simpler, although there may be performance issues (?).

AI

Something braindead to move towards the nearest opponent until within range, and then shoot. The AI could get additional position/velocity information from the physics engine to predict player movement for aiming, although there should probably still be some kind of random factor introduced (no aimbots, please).

Networking

The communication will probably use UDP, ticking along at some rate independent of the graphics FPS. Dealing with latency is the primary concern with a real-time game like Liero, particularly due to how weapons behave. To maximize local UI responsiveness, clients can do some form of client-side input prediction. To deal with unpredictable behavior of objects, clients might buffer incoming state and interpolate, at the expensive of increased latency. Dealing with latency in terms of aiming and weapon/player interactions is complicated, normal FPS lag-compensation methods mostly deal with hit-scan weapons, whereas Liero mostly deals with ballistic projectiles which can be dodged. The inherent network latency will be visible to the user in some form or another, so it may be better to just minimize the effect that the error has on the end results and deal with the initial latency. The protocol should handle packet loss in a sensible way, not retransmitting useless information, but also being able to deal with reliable events like player joins/chat.

Graphics

Something to represent the game state to the players. Characters and larger shots will be sprite-images. Resolution is most probably some constant. Ground and background are drawn with textures, given in a map-file, or random generated if the texture doesn't exist.

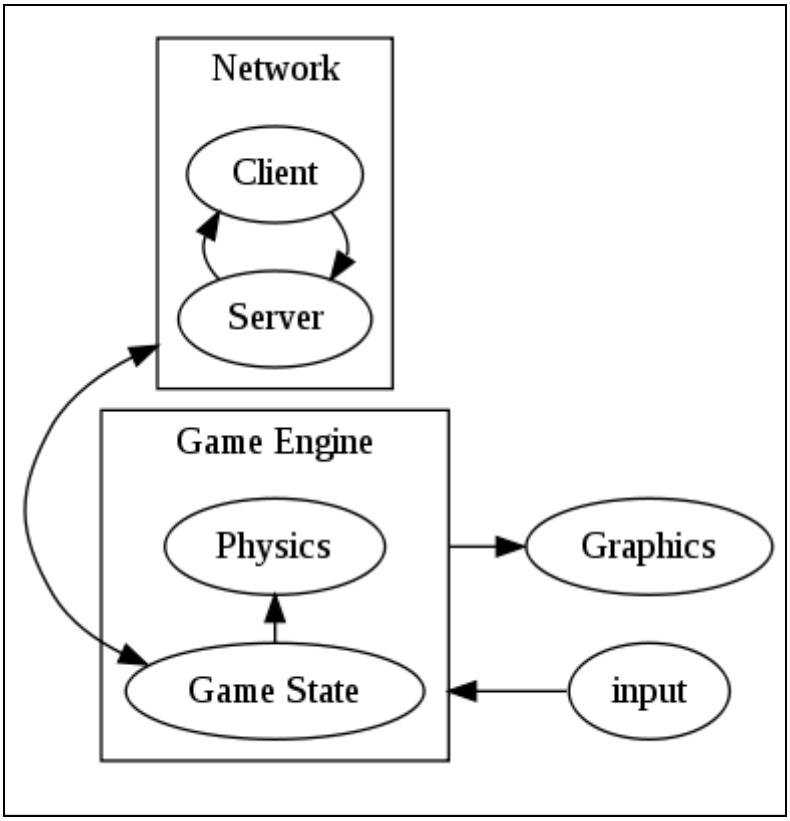
Auxiliary Components

Non-realtime-gameplay components such as the game menu, map editor, random map generation etc. are regarded as auxiliary components, and can be worried about after the core components are working.

Random Map Generation

Random walk algorithms/modified DFS could be used to generate a raw cave skeleton, which could then be smoothened out.

Diagram



Tasks sharing

Based on the architecture specified above, all team members must collaborate on the design of the core GameState structure and interfaces, but the other core components can then be separated:

Member	Core components	Auxiliary components
Tero	Network	Resources, Scripting
Marko	Graphics, UI (User Input)	Menus
Atle	Physics, AI	Random Map Generator
Eric	Physics, AI	Random Map Generator
Everybody	GameState	Documentation, Testing

Program testing

Since the GameState/physics components only deal with abstract data and have well-defined interfaces, they are well-suited for automated unit testing.

The network component can be tested using simulated latency/packet-loss.

Project schedule

Crazy Timetable

Week	Deadline	Task	Description
45	7.11	Project Plan	Discuss the design, write the project plan

45+	10.11?	Prototype	Implement the prototype
46	10-14.11	Plan Feedback	Receive feedback on plan
46	14.11	Design	Design the initial GameState and other core architecture
47+	24.11	Core components	Implement core components
48	1.12	Aux components	Implement auxiliary components
48	1.12	Core components+	Improve core components
47-48	1.12	Documentation	Write the documentation
49-50	1-12.12	Core+Aux components+	Improve core and auxiliary components
49-50	1-12.12	Demo	Demo working application

Eight Easy Steps to World Domination

1. Project Plan
    - ♦ Write this project plan

2. Prototype
    - ♦ A minimalistic, but working implementation of the whole GameState/network/graphics/user-input architecture
        - ◊ map = 800x640 box
        - ◊ players = 25x25px boxes
        - ◊ physics = gravity?
        - ◊ movement = up-down-left-right

3. Design
    - ♦ Design the initial GameState to be used to start implementing the core components
        - ◊ Storage of known/predicted states
        - ◊ Interfaces to G+UI/AI, Physics, Network components
        - ◊ Runtime (processes, threads, executables)
        - ◊ Object ownership (network server/network client/standalone client).

4. Core Components
    - ♦ Implement the Physics, Graphics, UI, AI, Network components to fufill the minimum requirements
        - ◊ Doesn't need a menu UI, can be operated using command-line arguments

5. Aux Components
    - ♦ Implement things like menu, map editor, etc.
        - ◊ GUI-polishing

6. Documentation
    - ♦ Write-as-you-go

7. ???

8. Profit!

References

- http://www.clanlib.org/ - ClanLib
- http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking - Networking
- http://www.gamasutra.com/features/20000511/bernier_01.htm - Networking Back-end services
- http://gafferongames.wordpress.com/game-physics/networked-physics/ - Examples of simple Networking
- http://www.ra.is/unlagged/ - Quake 3 lag compensation

- http://developer.valvesoftware.com/wiki/Lag_Compensation - Half-Life, Quake - Client side prediction, interpolation/extrapolation, lag compensation
- http://gafferongames.wordpress.com/game-physics/integration-basics/ - Integration Basics