Kishna Glista - Liero Clone Project Documentation

Atle Kivelä	79171V	atle.kivela@tkk.fi
Eric Malmi	80351A	eric.malmi@tkk.fi
Tero Marttila	79849E	tjmartti@cc.hut.fi
Marko Rasa	78726L	morasa@cc.hut.fi

December 9, 2008

Contents

1 Instructions for compiling and use

1.1 Configuring CMake

You must first generate the project's CMake configuration inside of *build*. A simple script is provided to do so with some default settings. It will configure the install path as *˜/opt*.

- *cd build*
- *./mkcmake.sh*

1.2 Compiling and Installing

Once you have the CMake scripts in place, compiling should be as simple as running make:

- *make*
- *make install*

This will build the binary, and then copy the binary and data files to the install path configured above.

1.3 Command-line Arguments

Running the game is done using command-line arguments to the executable

Short	Long	Value	Description	Default
-p	`--port`	PORT	Set server TCP/UDP port	9338
-s	`--server`		Run as a network server	false
-c	`--client`	SERVERHOST	Run as a network client on given server	false
-g	`--graphics`		Enable graphics rendering	*

* = true, except false when –server is given

The options `--server` and `--client` are mutually exclusive, and both cannot be selected at the same time.

1.4 Keyboard Controls

The default controls are identical to the origional Liero default controls, with some additions.

Action	**Default key(s)**
Move Left	Arrow Left
Move Right	Arrow Right
Aim Up	Arrow Up
Aim Down	Arror Down
Dig	Move Left + Move Right
Shoot	Right Control
Jump	Right Shift
Change Weapon	Enter + Arrow Left / Arrow Right
Throw Rope	Change Weapon + Jump
Release Rope	Jump
Change Rope Length	Change Weapon + Aim Up / Aim Down
Suicide	Left Control + K
Exit Game	Esc

1.5 Running the game

To simple start a local singleplayer game, just run `kg` without any arguments.

To start a network server on the default port, run `kg --server` (`kg -s`). You may optionally also specify the `--graphics` argument to have the server passively draw the graphics.

To start a network client, connecting to a server running on the default port, run `kg --client=ADDRESS` (`kg -c <address>`).

To use a non-default port, simply specify `--port=PORT` (`-p <port>`) on both client and server.

1.6 Configuration

All global game constants are defined in *src/Config.hh*, and may be experimented with. Weapon parameters are defined in *src/Weapons.cc*.

2 Program architecture

The program consists of four main parts: Graphics&Input, GameState, Network and Physics. Each part contains various classes; the relations show up in the doxygen generated documentation of the program.

The program starts from the Application class, which then starts the Engine, which creates GameState, Graphics and the Network Client/Server. Physics simulation is started when GameState is created.

GameState contains PhysicsWorld which inherits from Terrain and contains a list of PhysicsObjects. GameState also contains Player and Projectile objects which inherit from PhysicsObject (and are contained in PhysicsWorld).

Player objects have a list of Weapon objects which they can use to create Projectiles. Every Player also has a Rope object, which is a separate PhysicsObject, and is either folded away, being thrown or attached to the terrain.

Graphics and Input are handled in their own classes. Graphics has an Input object which contains InputHandler objects for various classes of input. One such object is PlayerInput which affects the GameState's LocalPlayer. Other object is GuiInput which just modifies what the GUI look like on the client side.

The network code is a bit complicated and has several layers. There are some nice diagrams about the program structure in the doxygen documentation.

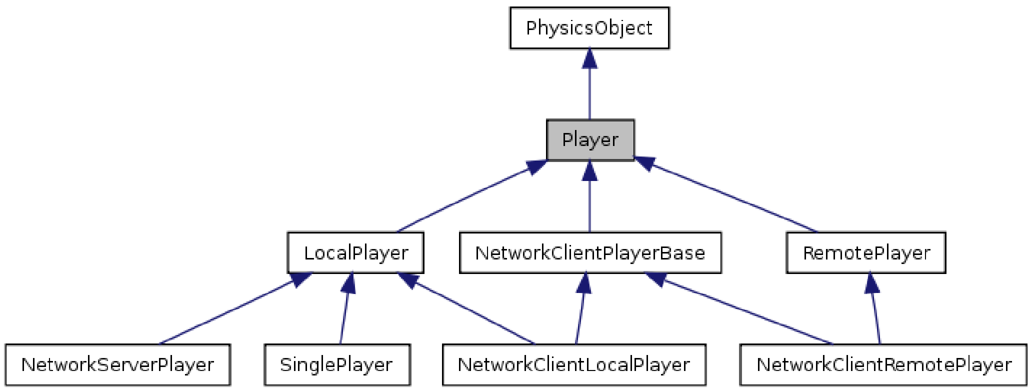

Figure 1: Player class inheritance graph.

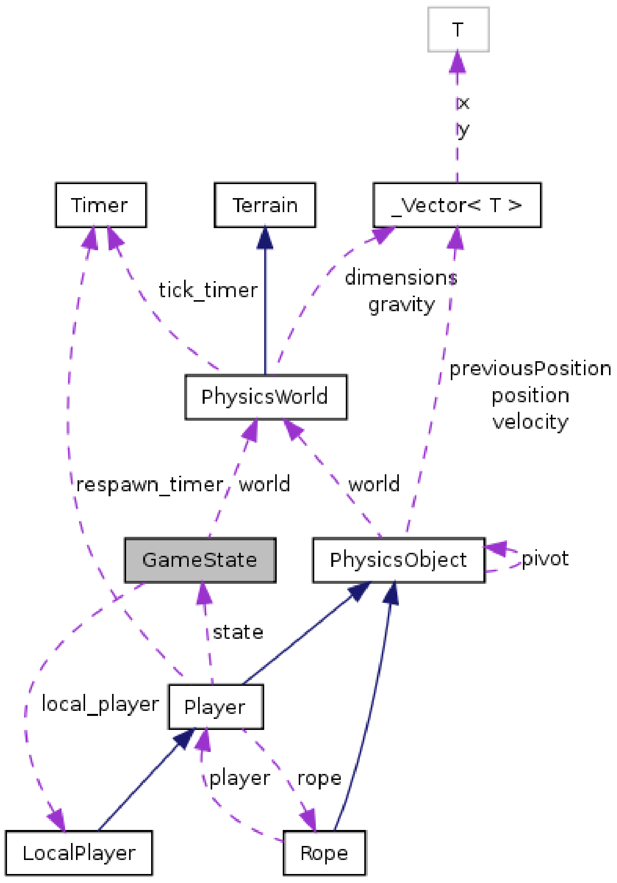

Figure 2: Relationships of core GameState class.

2.1 Network

The network code is implemented as separate NetworkServer and NetworkClient modules, which use a common high-level network interface, NetworkSession and NetworkObject.

The low-level details are implemented using ClanLib's CL_IPAddress (referred to as NetworkAddress) and CL_Socket. NetworkUDP provides an interface to send and receive NetworkPackets to/from specific NetworkAddress's across a NetworkSocket. NetworkTCP provides a NetworkTCPTransport interface, which can send/receive NetworkPackets on a NetworkSocket (using NetworkBuffer to buffer socket I/O). NetworkTCPServer is a listen() socket which accepts client connections as NetworkTCPTransports, and NetworkTCPClient is a NetworkTCPTransport that's connect()'d to some address.

NetworkSession encapsulates some simple application server/client behaviour, it can function as both a server, and represents remote NetworkSessions (either clients or servers) as NetworkNode objects. These then provide an interface to send and receive NetworkPackets on specific NetworkChannelDs, using either TCP or UDP as a reliable/unreliable transport.

NetworkObject then implements a kind of object-oriented network protocol. A NetworkObjectController (with specific subclasses for server/client behaviour) uses a NetworkSession to send messages on a specific NetworkChannelID. This controller then creates and looks up NetworkObjects (again, with specific subclasses for server/client behaviour). Clients and servers can then communicate by having the server construct new NetworkObjects (which are allocated an unique id), and then sending NetworkPackets with a specific NetworkMessageID type on a specific object. The message is then delivered directly to the NetworkObject instance on the remote end of the connection, or a new NetworkObject is constructed using the data in the NetworkPacket. This enables an easy way to send events for specific objects, and referr to other objects in these messages.

NetworkServer then implements a core NetworkServer class which has a NetworkSession and a NetworkObject_ServerController. Players that connect are represented as NetworkServerPlayers, which inherit from LocalPlayer and NetworkObject_Server. This class then overrides methods in Player to deliver messages on the Player object to the clients, or to create new NetworkServerProjectiles. NetworkServerProjectile inherits from Projectile and NetworkObject_Server, and sends messages when constructed, upon hitting a player, and upon being destroyed.

NetworkClient is a bit more complicated as it must handle both the LocalPlayer, and a number of RemotePlayers. Again, NetworkClient has a NetworkSession and a specialized NetworkClientController, which then creates objects of various other NetworkClientClasses upon receiving messages from the server.

Two of these classes are NetworkClientLocalPlayer and NetworkClientRemotePlayer. Both inherit from NetworkClientPlayerBase, which inherits Player (virtually) and NetworkObject_Client. NetworkClientLocalPlayer and NetworkClientRemotePlayer then also inherit LocalPlayer and Remote player virtually, respectively. NetworkClientPlayerBase contains the common methods that update the Player's state in response to messages received from the server. NetworkClientLocalPlayer overrides handleInput to send the input mask to the server, and NetworkClientRemotePlayer can handle remote clients disconnecting from the server.

In addition, there is a NetworkClientProjectile class, which inherits from

Projectile and NetworkObject_Client. this is created when a Player fires a Weapon on the server, and handles events received from the server like the projectile hitting a player (inflicting damage), or being destroyed (by hitting the terrain or something similar).

When the player first connects to the server, the server sends a large packet containing the terrain array to the client, which updates its own GameState world's terrain array with the received data.

Currently, the client only sends handleInput using unreliable UDP messages, and the server only sends position updates (as sent in response to handleInput events) unreliably. All other events are sent using reliable TCP.

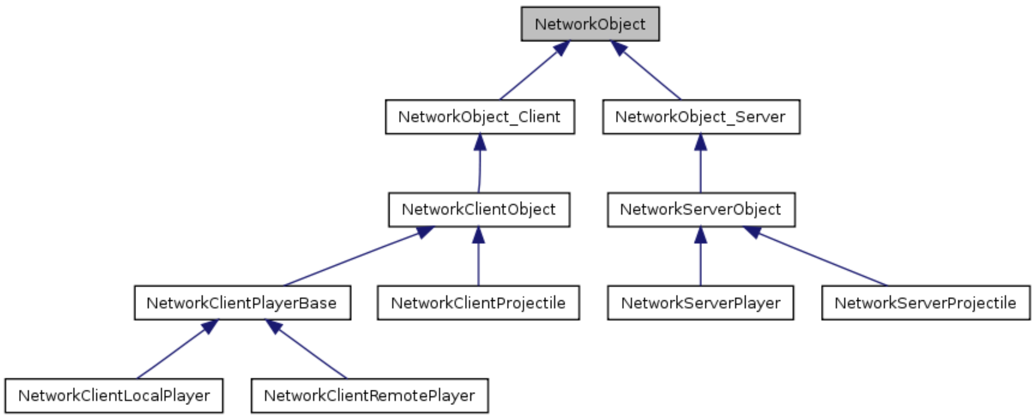

Figure 3: Use of NetworkObject in network code.

3 Data structures and algorithms

3.1 Basic data structures

- Vector
- List

3.2 World and Objects

The terrain is an important part of our game. We represent our terrain as a large array. Each shell of the array has a type that tells what is in that position. Currently, possible terrain types are EMPTY, DIRT and ROCK.

In our physics simulation the shapes of the different elements in the game are represented as polygons. A polygon is a vector of points that define the edges of the shape.

3.3 Collision Detection

- Polygon collision detection
- Pixel collision detection

Collision detection algorithms check if objects in the physics simulation are colliding with eachother. Because our terrain is represented as an array and objects are represented as polygons we have two different kinds collision detection algorithms.

The pixel based collision detection used to check collisions with the terrain is quite simple. It “draws” a line between two points A and B. The algorithm iterates over the line from point A to B and on each iteration it checks if theres a collision (i.e. the type of the current point is ROCK or DIRT).

In order to implement the bouncing from the terrain we have to be able to calculate a normal for the slope of the terrain in the collision point. We came up with the following algorithm which gives us an approximation of the normal

1. Take the collision point (p_1) and the point before the collision (p_2) and consider a 3x3 array of pixels which has the collision point in the middle of it.

2. Find the empty points that are connected to p_2 with bredth-first search algorithm or something similar.

3. Calculate the vectors pointing from the collision point to the empty points. Sum of these vectors gives us the approximation of the normal.

Picture 4 explains the algorithm a lot. In the middle of the pictured we have zoomed to the collision point. The red arrow is the sum of the black arrows and thus it is our approximation for the normal.

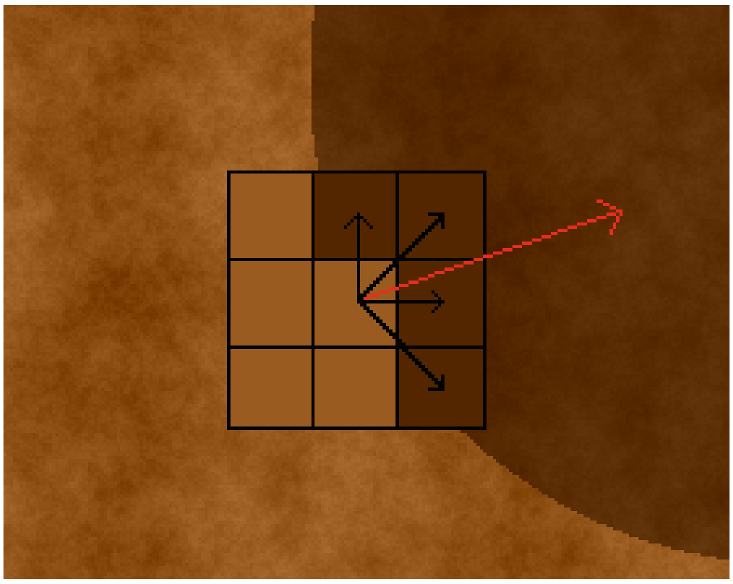

Figure 4: Visualizing the algorithm for approximating the normal.

3.4 Physics

The fourth-order Runge-Kutta method is a numerical method for approximating the solution of an ordinary differential equation. In our game the Runge-Kutta method is used to calculate positions and velocities of physics objects when we apply forces to them.

The mathematical formulation of the Runge-Kutta method: If we have an initial value problem of the form

$$y' = f(t, y), \quad y(t_0) = y_0.$$

The we can describe the RK4 method for this problem by equations

$$\begin{aligned} y_{n+1} &= y_n + \tfrac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\ t_{n+1} &= t_n + h \end{aligned}$$

where y_{n+1} is the RK4 approximation of $y(t_{n+1})$, and

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_1) \\ k_3 &= f(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_2) \\ k_4 &= f(t_n + h, y_n + hk_3) \end{aligned}$$

The next value $(y_n{+}1)$ is determined by the present value (y_n), the product of the interval (h) and an estimated slope that is defined as $\tfrac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right)$.

3.5 Texture generation

In texture generation we use random fractal terrain generation algorithm [2]. In one dimension algorithm starts with one straight line. Then it affects to that line's midpoint with some random multiplied with H-value. After those first steps it simply calls itself to both parts of line changing H-value smaller.

We use the two dimensional version of the algorithm. It needs step for point's between last density and step diagonal to last density's points – otherwise it is quite similar to the one dimensional version. We decide to implement the algorithm as iterative instead of recursion.

3.6 Input

All input is represented as a bitmask, composed of bits from *enum PlayerInputBits*. Additionally, the Graphics code uses some local-only flags defined in *enum GuiInputBits*. These bitmasks are built using the *InputHandler* class, which is defined as a generic template. On every update, it goes through its keymap, which is defined as an array of InputKeymapEntry structs. These contain the input bit, flags, and up to two keycodes. The InputHandler then reads keycodes from the keyboard and sets bits in the current input mask based on the entry keycodes (which may be negative to indicate the the specified key must NOT be pressed down) and any key-repetition rules defined by flags.

Key repetition is implemented using InputKeyRepeatQueue, which contains a list of InputKeyRepeatEntry's. To rate-limit keypresses, the input code is push()'d to the queue, and it eventually removed from the queue once it expires, or the key is released.

The Graphics code then reads the input mask from time to time, resetting the InputHandler's mask to zero, and passes it on to LocalPlayer, which then either handles it locally, or sends it to the remote server.

Since some inputs like walking, aiming and moving up and down the rope are time-dependant, the InputHandler also tracks how many milliseconds the input mask has been held, and this time delta is applied by LocalPlayer.

4 Known bugs

1. If player dies while rope is attached the rope will still be attached when the player spawns.
2. If rope is thrown without releasing it first, rope will pull worm while in mid-air
3. Collisions with the terrain are only tested for the vertices of the polygon. It is thus possible for the player to move through some pixels.
4. Existing Player ropes and Projectiles are not sent to the client when it connects, which can cause apparent glitches in what the terrain looks like and how players move.

5 Tasks sharing and schedule

We could have followed the schedule a lot better. We basically forgot the whole schedule and had a lapse in activity during the middle weeks, which caused us to be delayed in terms of the schedule. The positive side was that we almost always had all the team members working on their own things in parralel and communicating together; either at Maari or using our IRC channel.

Tasks sharing worked pretty much as planned. Tero did all the network code and worked on keeping the rest of the code network-safe. Most of our eye-candy (like terrain textures) was done by Marko, who was responsible for the graphics. Marko, Eric and Atle worked on everything Physics related plus the

GameState/Player/Rope/etc code. Most of the time all team members were working together, so the code was written using common agreement.

We feel that the workload was shared reasonably evenly.

6 Differences to the original plan

The original plan was quite loose and it let us make decisions during development, which was a good thing. The basic structure of the program is pretty much as the one we thought about while planning, although the Network code ended up being a fair bit more simple-minded due to lack of time to implement more UDP-based behaviour.

Currently, the program lacks an AI which was in the original plan and most of the optional features were also omitted. Liero clone proved to be, however, a very interesting software project and we are hoping to release Kishna Glista 2.0 one day.

References

[1] Gaffer on games. Game Physics. 2006.
http://gafferongames.wordpress.com/game-physics/ (read: 2008-12-08)

[2] Terrain texture generation
http://www.gameprogrammer.com/fractal.html