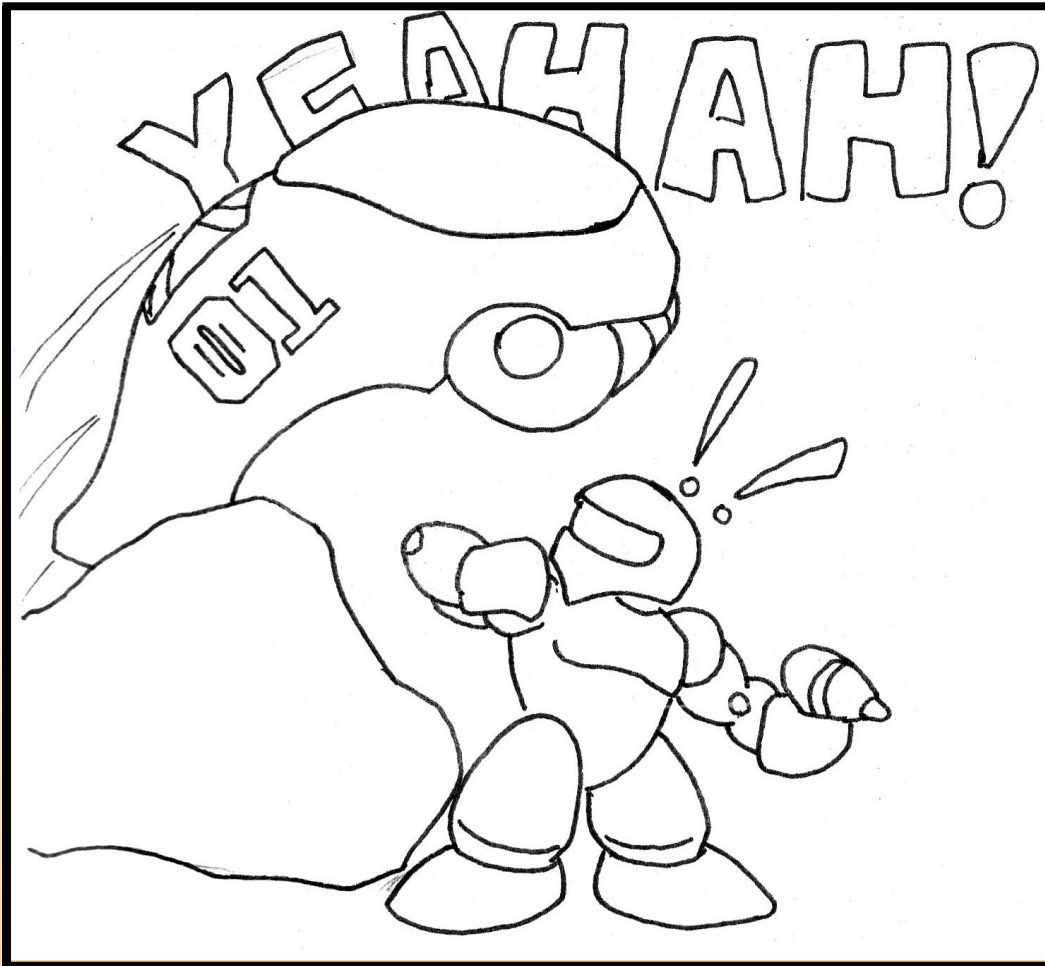


KRATERBIKES K

Ian Andersen :: Nicholas Retallack :: David Lock :: Thomas Gerdes :: Kunal Arya :: James Fallon





- Live Action!
- Multiplayer!
- Death Match!
- In Space!
- With biKes!

Summary of Presentation

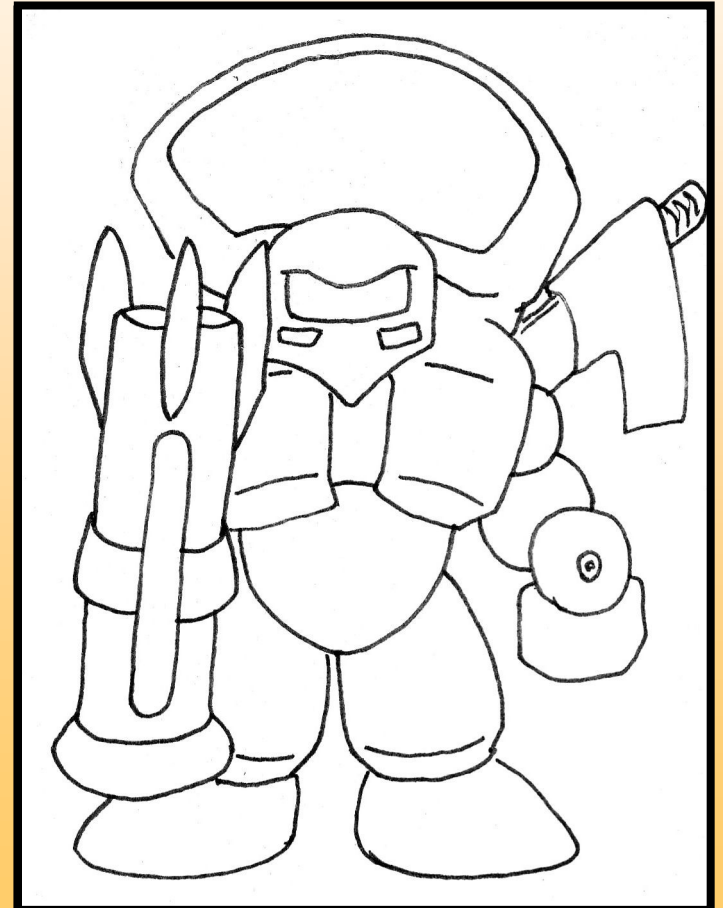
- . Overview - James
- . High Level Architecture - Thom
- . Detailed Design
 - . David – Physics and Game Engine
 - . Kunal – Network
 - . Nick – Input
 - . Thom – Rendering
 - . Ian – Sound

. **What it is:**

- . Multiplayer space shooter
- . Death match game-play
- . Features a sweet physics engine that incorporates collision detection and gravity
- . Multi-Planet playing field
- . Movement on planets and in space

. **Wish List**

- . Projectiles
- . Fighting On Foot,
- . Fighting In Bikes



· **Current Situation**

- 95% done with the core gaming features based on our requirements
- Working on polishing this core component
- Working on extra features at this time
- There is no guarantee that we will get these extra features done

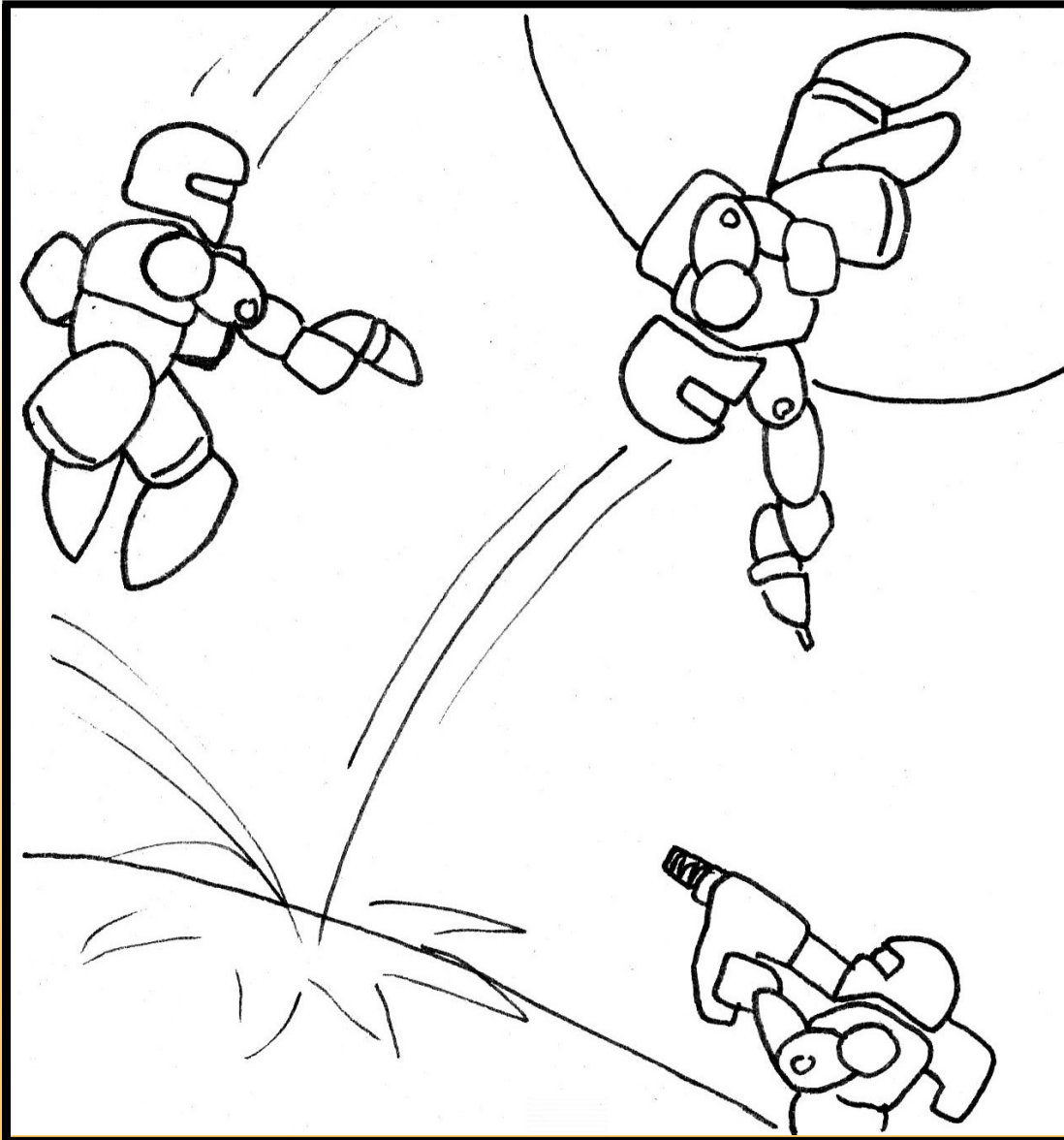
· **Possible Project Risks**

- None concerning the core of the game
- Loss of dedication to the project to complete “Extra Features”

· **Project Complexity**

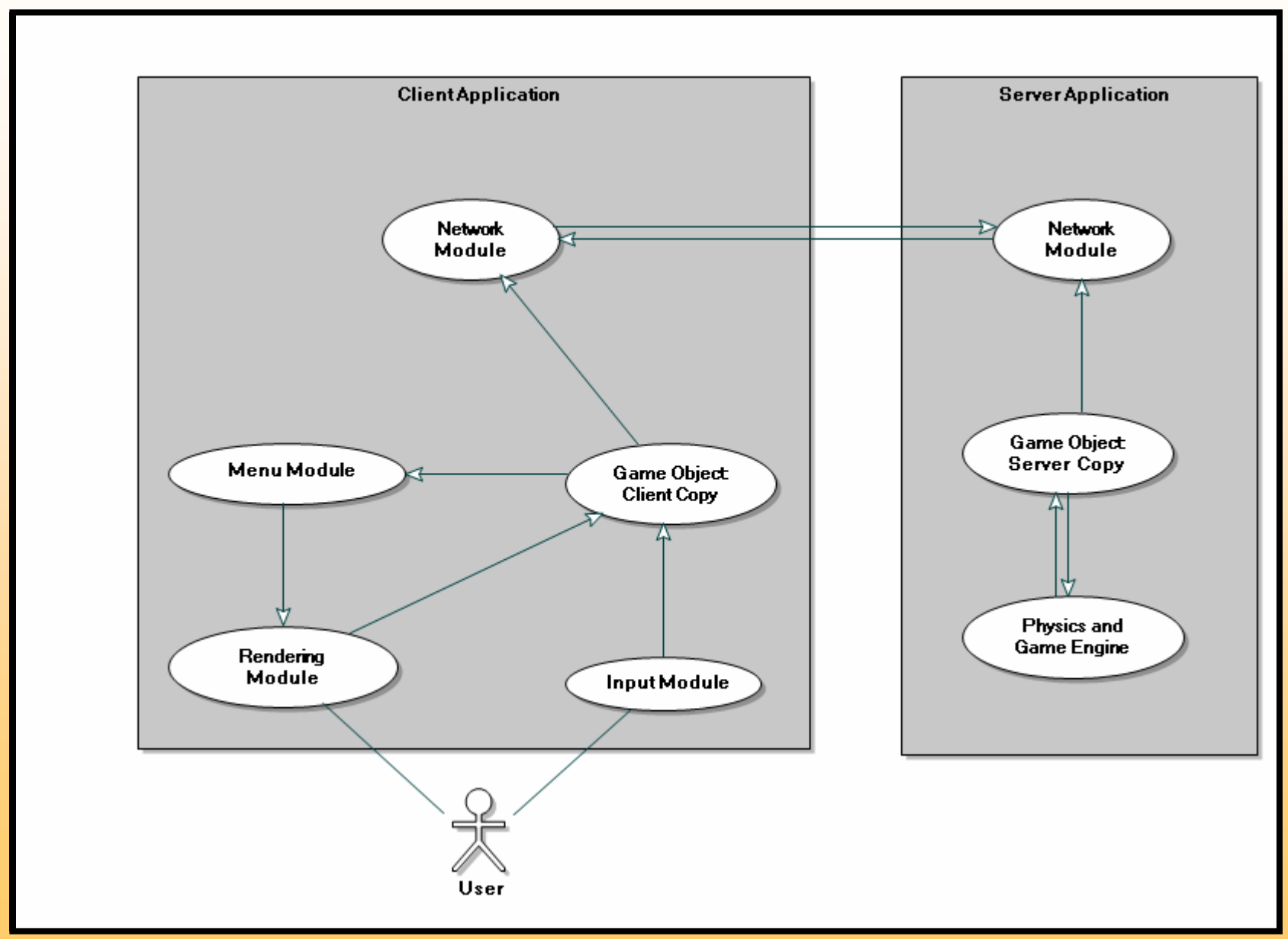
- Very high, however we kept the modules and interfaces as simple as possible, therefore increasing productivity and efficiency.

Game Play

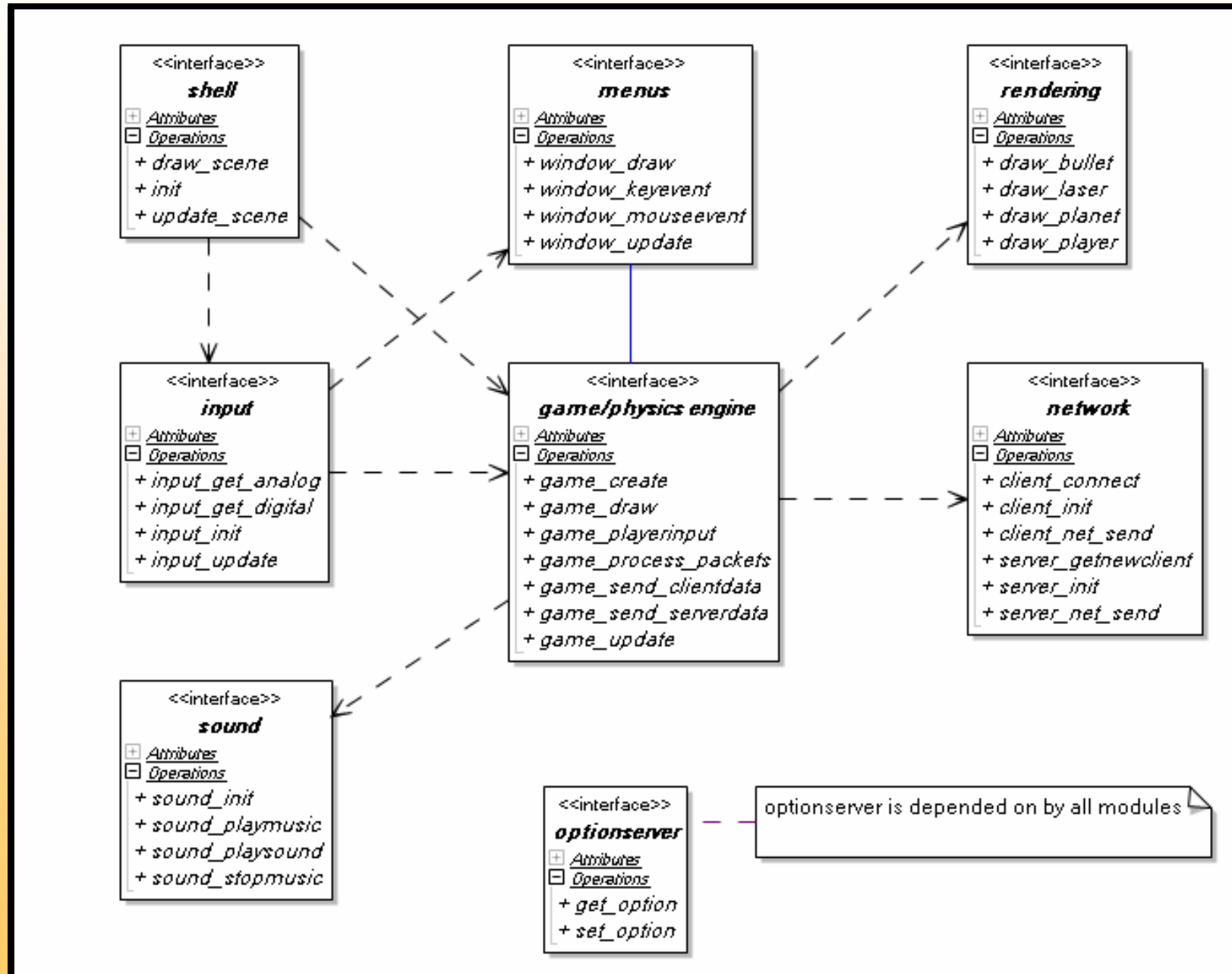


- **Deathmatch**
 - Kills counted, highest kill rate wins.
- **Movement**
 - Forward/Backward movement on planets.
 - Thrust/Turn/Reverse movement in space.
- **Attacking**
 - Mouse controls an aiming reticule that follows a circular path.

Client / Server Overview

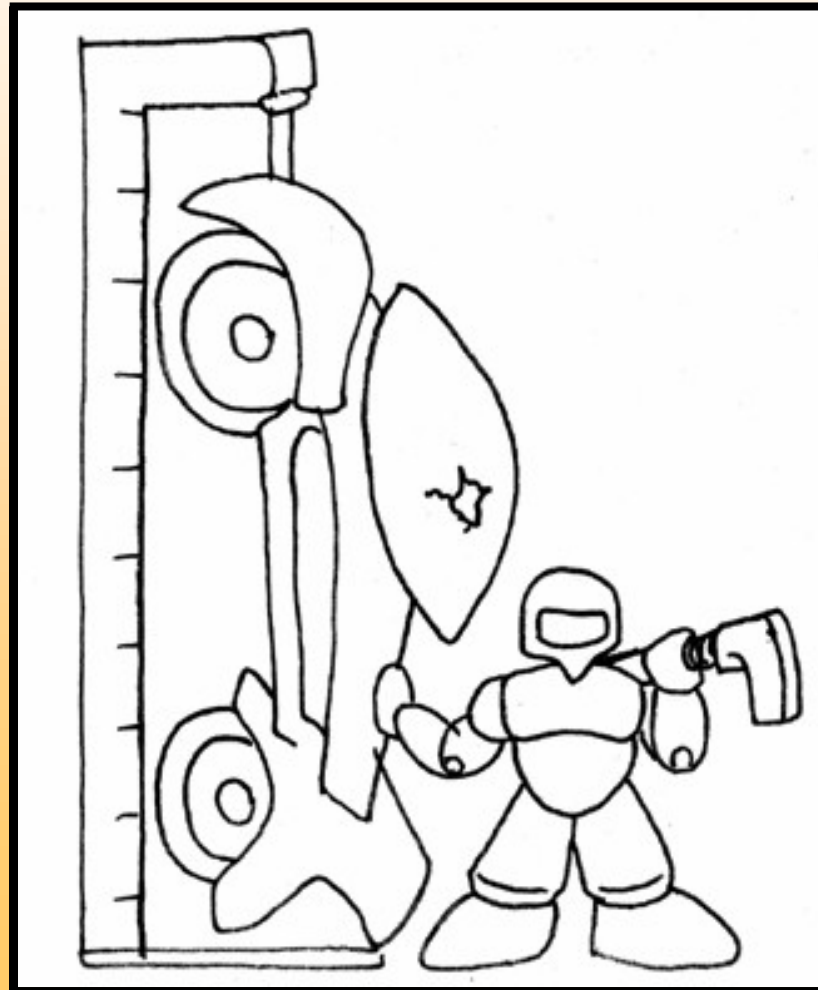


High Level Architecture



Physics and Game State

Overview of Game state data and physics calculations needed for KraterBikes

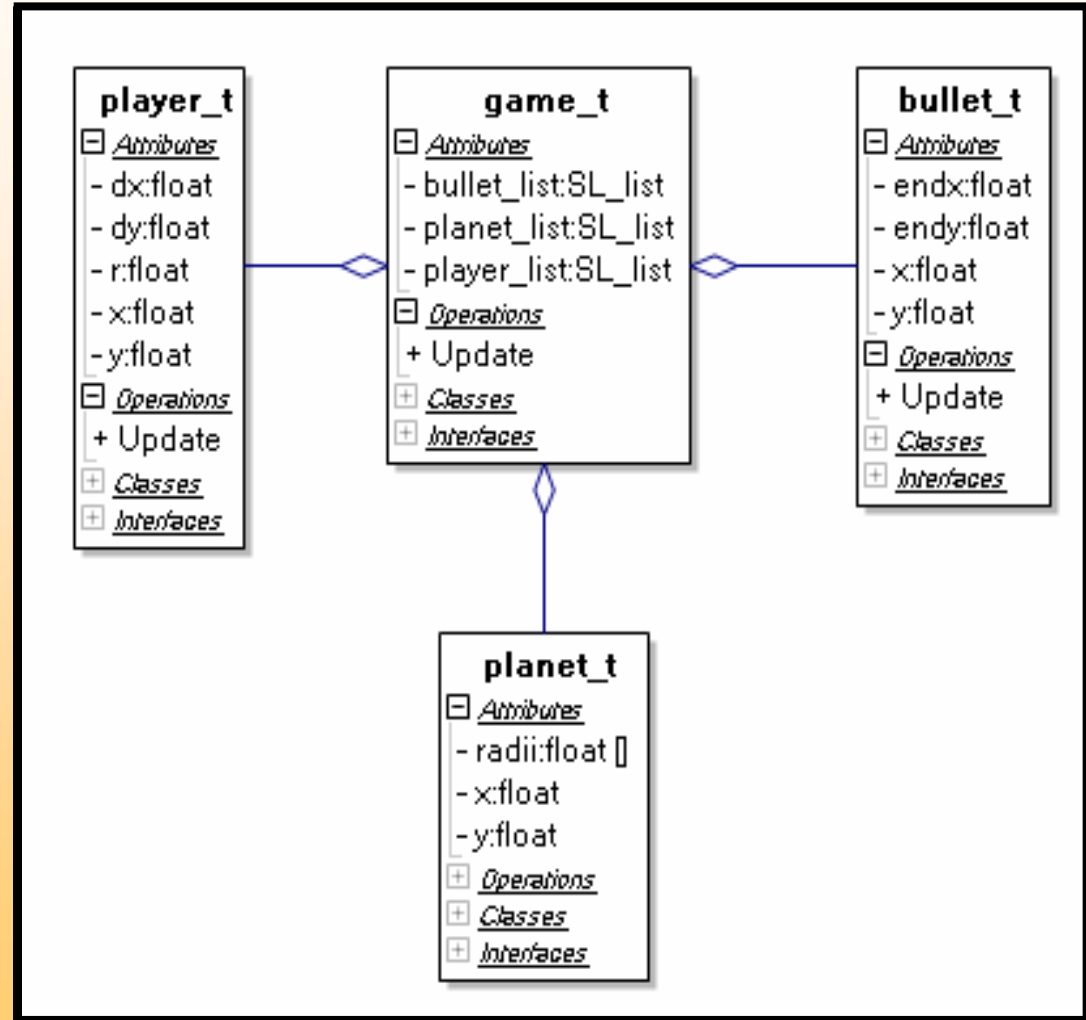


Overview

- **Data**
 - Player Position
 - Player Controls
 - Planet Information
 - Bullet Information
 - Chat Messages
- **Functionality**
 - Send Network Info
 - Update a clients game state to match the servers
 - Update Bullets
 - Calculate bullets hitting planets
 - Update Players
 - Calculate player vs planet collisions
 - Calculate player vs bullet collisions
 - Move Players on planets and in space

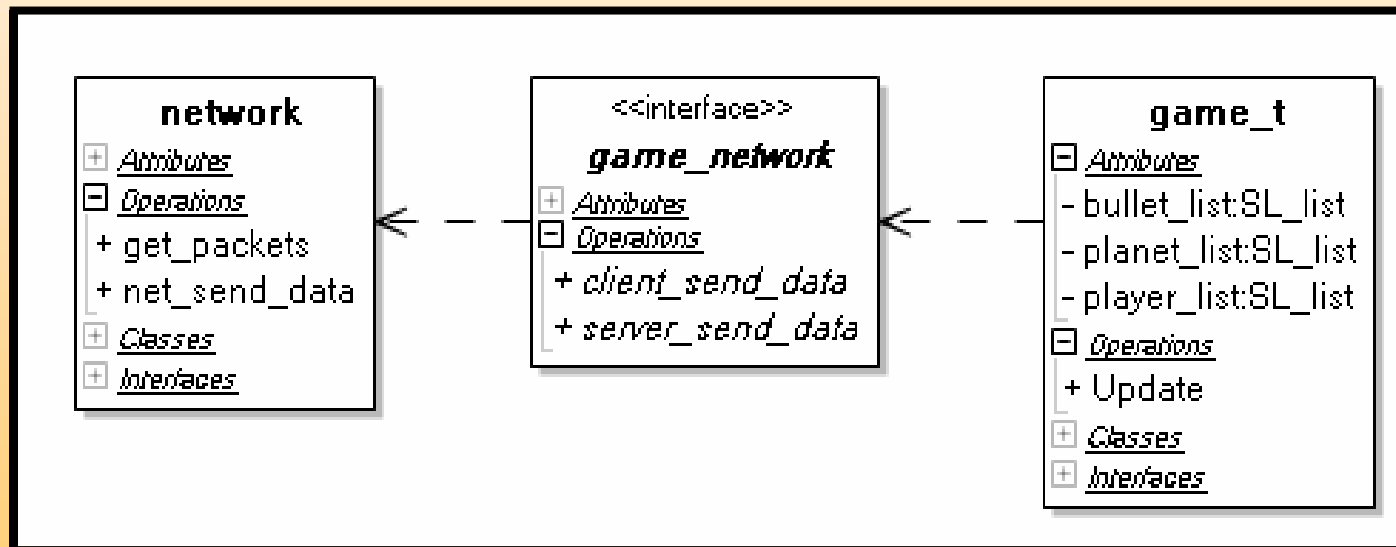
Basic Structure

- Simplicity is key
- Update functions perform physics and game logic
- Game_t recurses the update call into all players and bullets it owns
- game_t's update called once per frame



Network Updates

- Keep copies of game state on client and server
 - Server has master copy
 - Client collects user input and sends to server with `client_send_data`
 - Server updates game state with `Update` and sends to clients with `server_send_data`
 - Clients render the update game state for the user

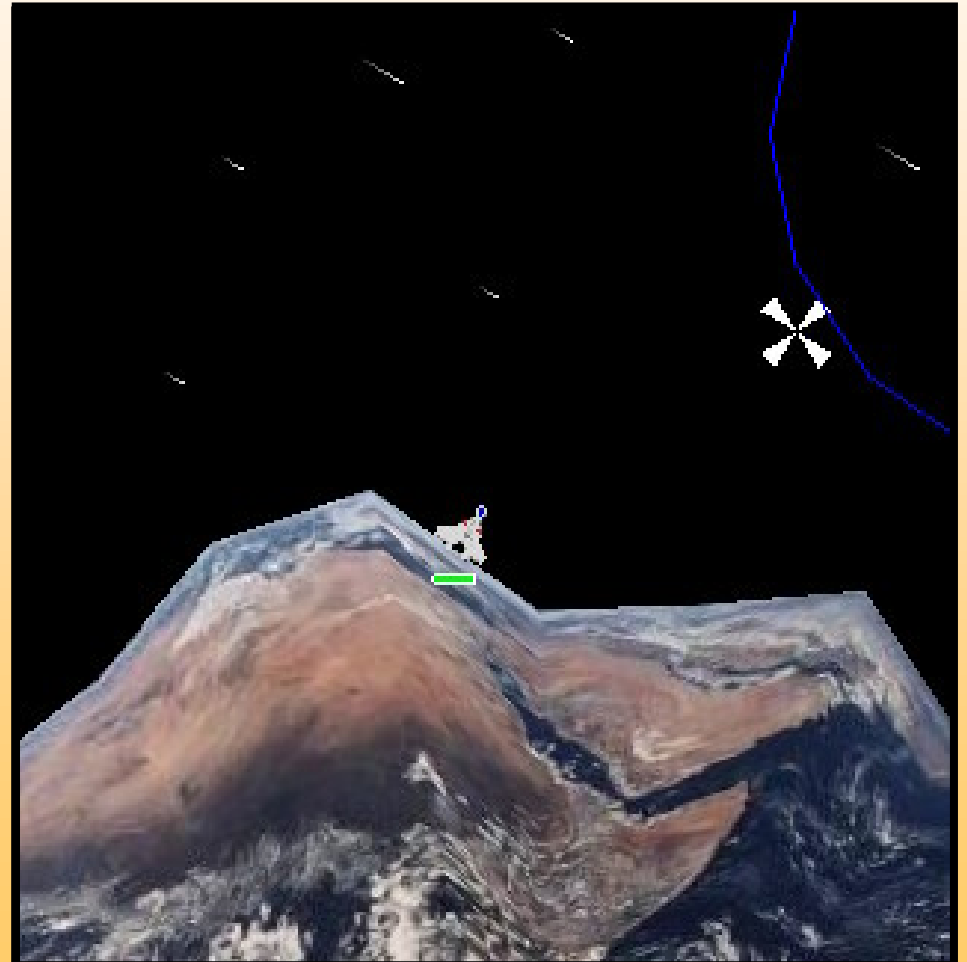


Interface : Simplistic Design

- **Game Engine exposes three main functions**
 - `game_update()` to update the game state on server
 - `game_create()` to create a new game
 - `game_render()` to render the game to the clients screen
 - `game_playerinput()` to collect player input
- **Game Network interface exposes only three functions**
 - `send_client_data()` to send basic client data
 - `send_server_data()` to send basic server data
 - `process_packets()` to process incoming packets

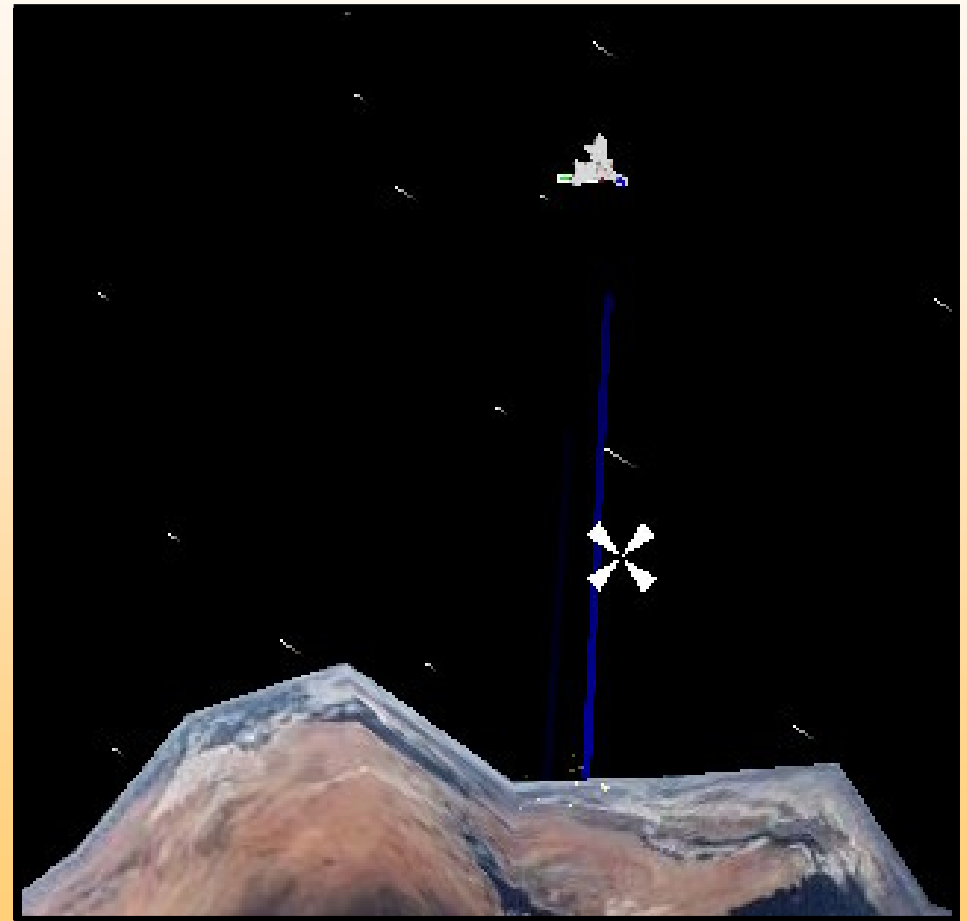
Physics Overview: Player vs planet

- Vectors which make up the planets surface calculated
- If player is below one of these vectors
 - Player is moved to the surface
 - Player velocity is calculated for reflection and friction



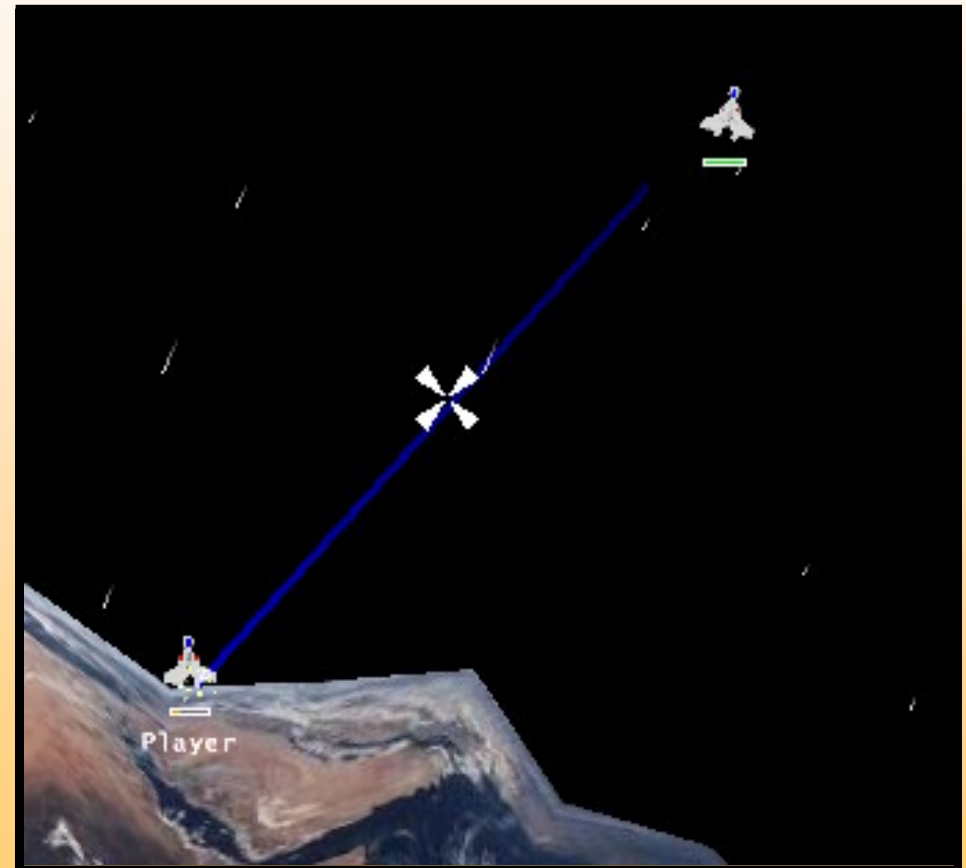
Physics Overview: Laser vs Planet

- Planets vectors again calculated
- If laser vector hits planet surface vectors
 - Laser path is shortened to outside of planet
- Perform all planet checks before player vs laser checks



Physics Overview: Laser vs Player

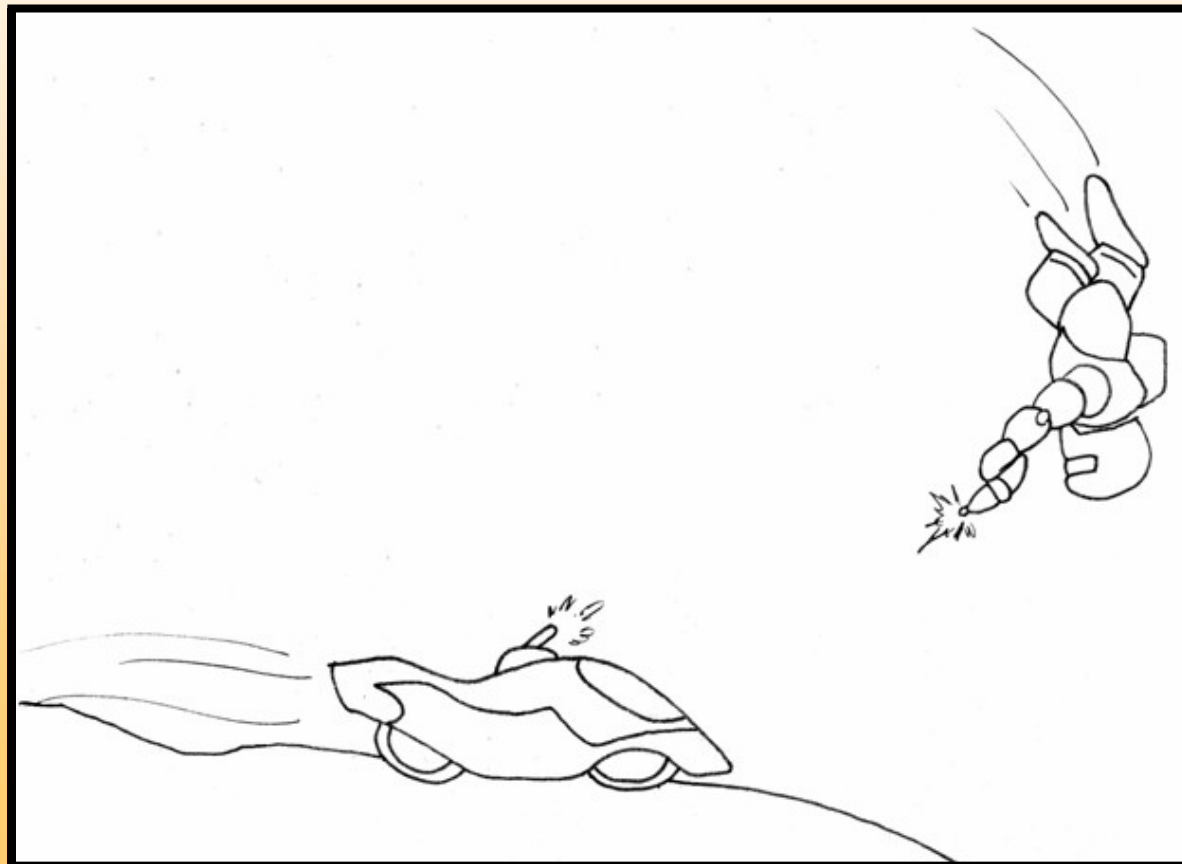
- Laser vector calculated against a bounding circle centered on the ship
 - If laser hits ship players health is reduced, and is knocked back



Physics Overview : Tiered Detection

- Collision detection is performed in tiers, so simple fast functions are always ran first, to prevent more difficult functions from having to run if the collision is clearly not happening
 - Laser vs player checks to make sure player is in the direction of the shot
 - Player vs planet does a simple distance calculation on the maximum radius of the planet
 - Laser vs planet does a line vs circle calculation on the maximum radius of the planet

Networking and Threading



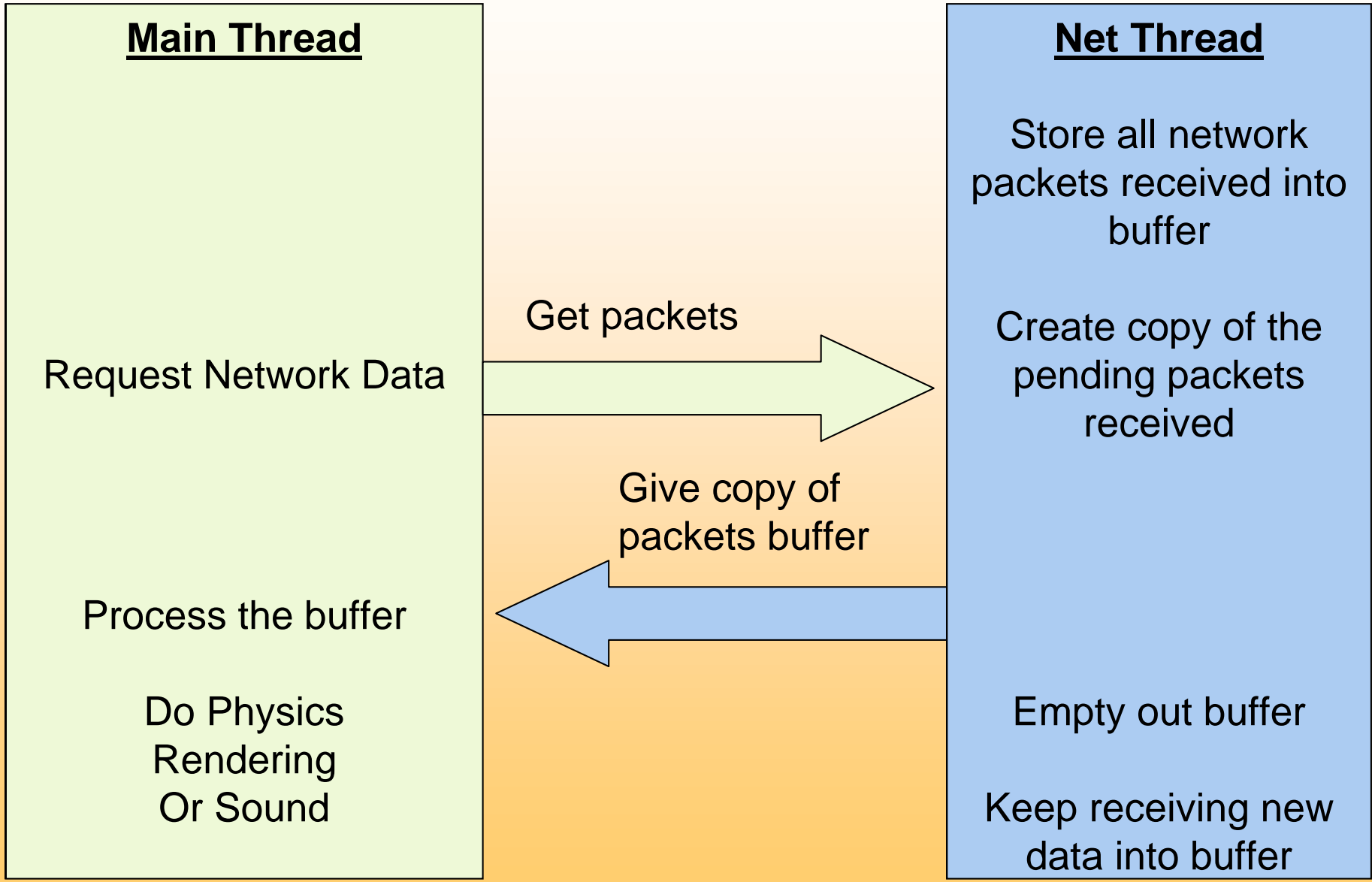
Requirements

- **Real-time communication of game data over network**
 - Ideally want 24 frames per second
- **Reliable data packets**
 - Ensure all data is received
 - Lets game engine decide what to keep and what not to keep
- **Low CPU usage**
- **Asynchronous**
 - Main thread can render while getting network packets
- **Arbitrarily sized**
 - For allowing features such as chat

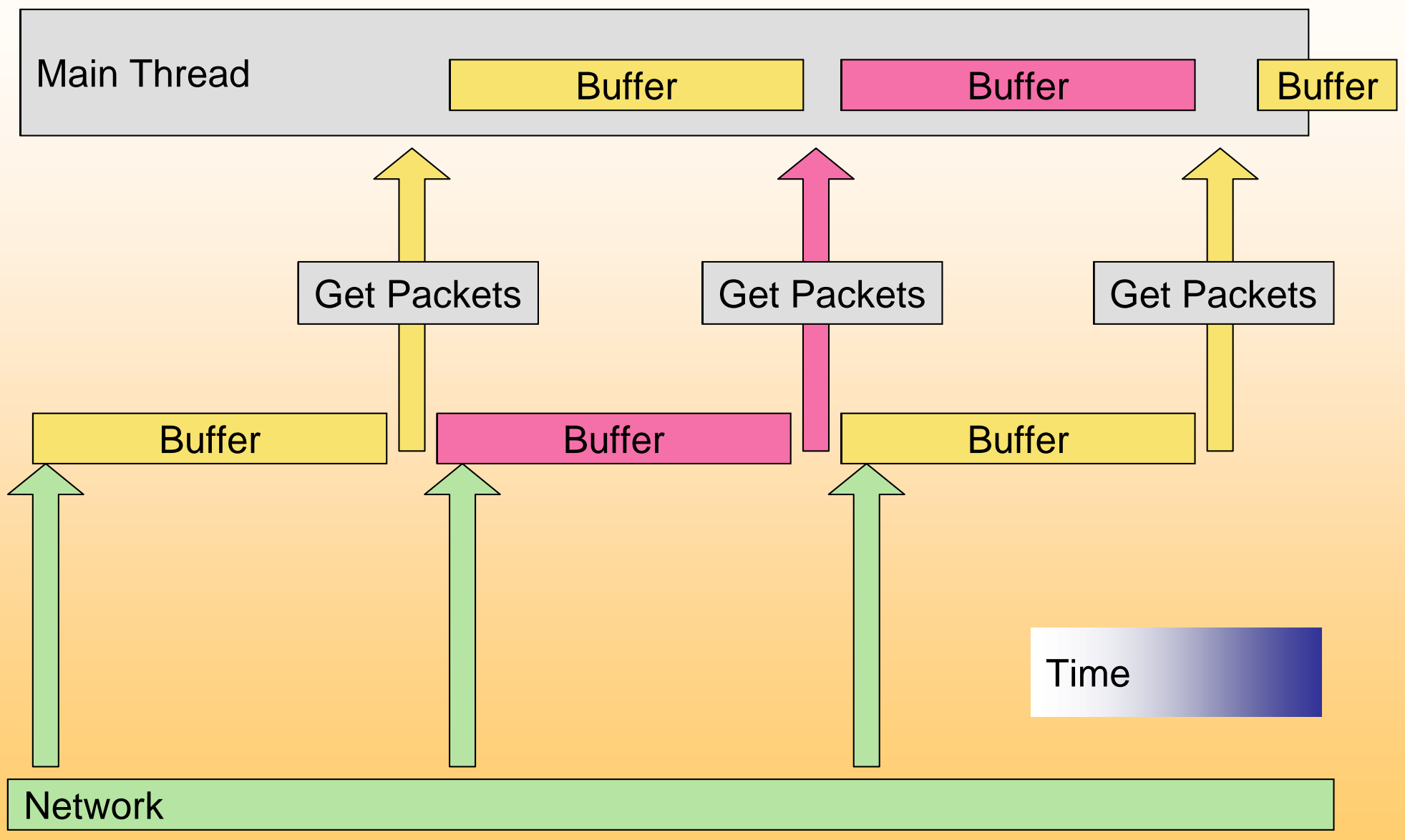
Threading

- Need a thread for each socket open
- Allows for asynchronous data transfer
 - Thread receives data while game engine does other things – rendering, physics
 - When the game engine requests data, it gives all data received since last request

Data Flow



Process

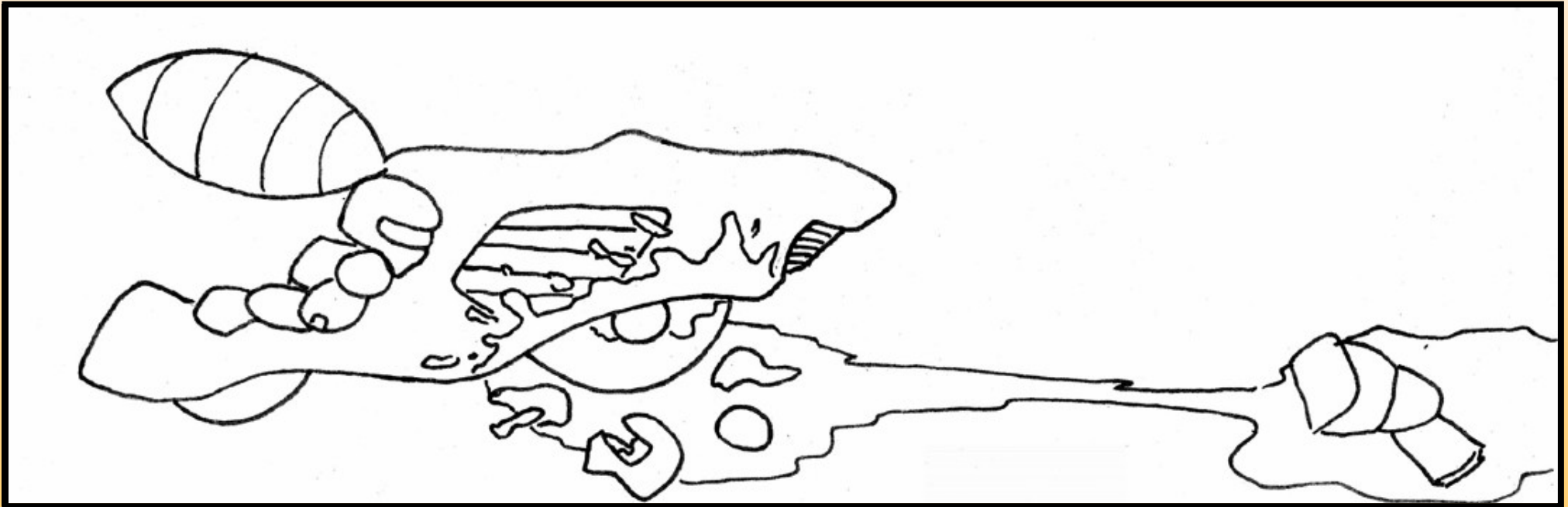


Performance

- During tests, the code can handle ~600-700 KB/sec per client @ 8 clients
- Only limited by processing power
- Data transfer test program
 - Data integrity using primitive checksum
 - Speed
 - CPU Usage

User Input

- Supports Keyboard, Mouse, and Joysticks
- Player actions can be remapped in a ridiculous number of ways!
- Includes Text Editing



User Input

- Supports Keyboard, Mouse, and Joysticks
- Player actions can be remapped in a ridiculous number of ways!
- Includes Text Editing

Interface

- **input_init:**
 - Allocates data for devices and mappings, and turns on SDL event handling states.
- **input_update:**
 - Updates the device states, and uses them to update the player's actions.
- **input_get:**
 - Allows access to all player actions, menu commands, cursor positions, etc.
- **input_free:**
 - Cleans up allocated data
- **typedef input_accessor_enum**
 - provides all constants that can be passed to the input_get command
- **Also provides miscellaneous debugging routines.**

Structure

- **Interface**
 - maintains local copies of Device and Mapping objects.
 - Routes general requests to the module they apply to.
- **Devices**
 - Maintains local states of currently attached devices, by iterating through input events.
 - Provides interface with access to core signals.
- **Mappings**
 - Loads or saves an input mapping set
 - Applies an input mapping set to the devices object passed to it to determine the player's actions.
 - Provides interface with access to player actions.

Devices

- **Updates internal device states**
 - mouse, keyboard, and multiple joysticks
- **Edits the current text buffer**
 - full ascii support
 - allows cursor movement, backspace, insert, and delete
- **Maintains menu signals**
 - provides accessors to check if the user has clicked an where, if they're done editing text, where the cursor is, etc.

Mappings

- **Loads and Saves input mapping files**
- **Updates player actions, and provides access to them**
 - Iterates a list of mappings
 - Maps half-joystick-axes to analog or digital commands using deadzones
 - Maps buttons to digital or analog commands by setting the sensitivity they generate
 - Allows multiple extra methods of aiming.

Risks and Complexity

- **Extreme complexity, with a simple interface:**
 - I like to take on difficult tasks, so I designed the interface to make anything possible.
 - Exports data in nearly the same form as is used by the network, so no changes have to be made outside my module if I change my mind.
 - Ridiculous possibilities for input mapping, because I can. Though it was somewhat painful at times.
- **No risk, because it works already:**
 - I have already provided the team with an input module that does everything we need. What I'm working on now is an improved design of it.

- .Texture Loading using SDL_image
 - .Supports Jpeg, png, gif
- .Open GL for Rendering
- .game_t holds all objects to be rendered
 - .game_draw
 - .starfield_draw
 - .planet_draw
 - .player_draw
 - .laser_draw
 - .message_draw



Sound

- The sound module exposes the ability to play a sound effect, and start music. This is all that is needed by the game engine to play the appropriate music and sound for the players at all times.
- Very simple to implement and easy to port to other projects.
- Uses the sound component of SDL called SDL_mixer. The mixer is needed to play multiple sound files at once.
- module simply polls for the position of a player based on his (x,y) position and plays music accordingly.

