The Quarks: A Game Programming Project Wrap-Up

Ian Graham & Nick Carter

13th December 2002

Overview

The Quarks is a networked, multiplayer, team-based capture-the-flag game. Each team controls a band of characters called Quarks–one player per character. The object is as that of standard capture-the-flag: each team has a base and a team flag, and the objective of the game is to capture the opponents' flag and return it to your base. Each Quark is able to walk and jump small distances, as well as perform special abilities which allow them to modify the level terrain as well as directly interfere with other characters:

Rocket: Rockets fire across space and explode on contact, obliterating terrain and damaging Quarks caught within their blast radius.

StickyBrick: Bricks may be chucked at nearby walls–these StickyBricks attach to the first piece of terrain they hit, expanding to create a region of brick with some radius.

Variable blast radius: Both types of projectiles may have different radii of effect upon impact. The radius of a projectile is increased by charging up the firing Quark over time before releasing the projectile.

Quarks come in several flavours, including Up and Down.Effort was taken to assure that controls were easy to use and relatively simple. More than 95% of gameplay is accomplished via a three-button mouse. Mouse motion controls a cursor that scrolls the level in world space. Mouse clicks direct a character to perform an action – the left button to fire a rocket, the middle button to fire a stickybrick, and the right button to walk and jump (which are performed as a unified action made possible by path planning). Clicking and holding the left or middle buttons results in charging up a projectile so as to increase the size of its blast.

Development Summary

Structure of code:

Code was written with design effort placed in a few categories, including (a) thoroughly data-driven design, (b) mindfulness of correctness in the client/server model (c) reusability, via class abstraction, of code not inextricably tied to game logic. Furthermore, an interactive framerate of at least 60Hz was regarded as necessary. Lastly – and most importantly – the game was to be fun; as designers we were careful not to let techincal features overshadow the enjoyable aspects of gameplay. The code structure can be outlined as follows:

1. Map Renderer. A class module designed to encapsulate the dynamic terrain model of our game. Its implementation was based on a quadtree. Functions to efficiently test for intersection with this terrain, as well as render and modify it, were provided as methods to the class.

2. Resource Managers. Three resource management modules were deemed necessary for consolidated management of memory-intensive data. These were a texture manager, a model manager, and a sound manager. The purpose of these classes was to allow separate code sections to share the same data instance without explicity knowledge of each other. Each was implemented as a singleton thatprovided namespace abstraction (which, in the case of textures and sounds, was a mapping from a logical namespace to physical path locationss where data might be found.

3. Pathfinding Module. A pathfinding module was created to supply a unified means for character movement and navigation. The primary search was A*, with several modifications made for efficiency.

4. Network Code. TCP/IP socket connections were virtualized by writing an application-level protocol layer that, at its most fundamental layer, was able to marshal and unmarshal C-language structs lacking pointer indirection. Care was taken to remain independant of machine byte order, and the code has tested successfully for communicationg between big- and little-endian machines.

5. Centralized Game State. To the greatest extent possible, efforts were made to avoid reduplication of code across client and server modules. Where possible, game state was represented by a common class that both could use in the same manner. This approach was also used along with carefully-crafted network code in order to prevent synchronization issues both between client and server and between multiple clients.

6. User Interface. Existing within the client module was code to provide a usable and enjoyable user interface that fully exposed the abilities of the

game engine. Examples include mouse and keyboard input, a carefully-select color-coded indicator system for both onscreen and offscreen characters, a pretty startup screen, and score display for individual characters and teams overall.

7. Eye Candy. Care was taken to provide visual effects such as lit, multi-textured terrain, explosions (through the infamous Exploder class. Exploder!), motion-blurred and real-time deformable flag animation, and smoothly interpolated character animation sequences.

In addition to the bitchin' work just described, our project contains numerous violations of such national and international copyright laws as are typically held in esteem by the legal community.

Borrowed Code

1. an OpenGL MD3 loader, adapted from that available at planetquake.com
2. libjpeg, libtiff, libtarga; for image loading. We rolled our own texture management code. Hyarr.
3. FMOD for sound.
4. Our bugfixed version of A. Willmott's SVL, which was used only sparingly, as most math turned out to be integer-based.
5. Lib3ds, which we wound up using not at all, but which remains nonetheless in our source tree.
6. Borrowed Content
7. Textures, from various web sources, and Al Reed
8. MD3 models, from planetquake.com
9. MP3 music, from various artists
10. Sound effects from various sources

Three Key Technical Challenges

1. *Managing game state across a network.* Accomplished by an effective design which gave the server the task of being the synchronizer of last resort while providing clients the flexibility to independently target character actions in the event that server data was not immediately available. This design turned out to be a reasonable tradeoff between ease of coding, latency, and quality of rendered output.

2. *Character navigation.* Managing path planning and physics across a changing terrain–a broad, modular path system was created and used to manage most character movement. Care was taken with regard to state synchronization, efficiency, and game balance–a particular concern was that pathfinding is easy to make powerful enough to detract from game challenge. Measures to ensure determinism, clever data structure management, and path limitations (caps to distance and number of nodes) were used to address these concerns.

3. *Dynamic terrain.* The requirements of the game called for a system that could support additions and deletions of terrain in regular volumes. Early on this was translated into quantizing the terrain and having all additions and deletions take place on some smallest indivisible scale. We represented this using a full quadtree rendered as a seamless triangle mesh. Much difficulty was encountered eliminating T-joints from the mesh, and rendering it in a reasonably fast way. Ultimately, texture-order traversal was selected.

Postmortem

Three (+1) things that went right

1. *Application-level, packet-based network I/O.* Our system would pre-buffer data frames from the network stream, and provide parsing and unparsing in an automated way. At critical times, this allowed us to modify our protocol without having to re-write I/O functions.

2. *Weekly planning meetings and explicit coordination.* We established a discipline of meeting regularly – at least one formal meeting per week. At these sessions we reviewed status and provided short-term goals within the context of larger project milestones; also, labor was divided and balanced.

3. *Mutual motivation.* Both partners were mutually motivated by a project design which was interesting to them both. This was largely due to good communication early on, in the planning phases, which caused the project to be something that both partners felt a sense of ownership for. (4)

4. *Tight, flexible code; code of the sort upon which you might expect to have a meal served your mother.* Good coding skills and modularity allowed us to completely repurpose our game engine (from a race-to-a-goal game to a capture-the-flag game) in the three hours just before the deadline – proof that foresight can pay off in the long run.

Things that went wrong

1. *Not anticipating that our original game design wasn't going to be fun.* We became too caught up in enjoying playing around with the technology

we'd created that we lost sight our game objective. Our original design failed to provide much interaction between players – typically, only when they passed each other at a single halfway point while following opposite paths. Earlier prototyping would have eliminated the need for last-minute repurposing.

2. *Animation overkill.* Our original plan called for a skeletal animation package using lib3ds. This was overkill, as our project requirements only included the playback of a small set of animations, and no dynamic skeletal animations. It was a mistake to not realize this at the outset – the advice against building swiss army knives rings true

3. *Reliance on available content.* We ought to have had a more comprehensive approach to designing a game and storyline. We let available content largely determine our game's appearance. Having at the outset a stronger idea of what's desired would have been more efficient and probably led to a higher quality product.