Quake Engine Analysis

Zachary Hickman

Northeastern University

zhickman@ccs.neu.edu

Introduction

In 1996, id Software published *Quake*, a first-person shooter video game that succeeded the company's previous hit, *Doom*. It was the first FPS published by the company to be fully 3D, and added online multi-player support. The impressive feature improvements required a much more comprehensive engine.

This analysis focuses on the major sections of *Quake*'s engine, with special detail in the two most technologically advanced features at the time: the network functionality, and the rendering/drawing systems.

The source code was analyzed at id Software's official GitHub page[6].

I. Time and the Game Loop

In designing the architecture of a game engine, defining what sort of time the game loop is based on – game time, real time, CPU cycles, etc. - is critical, as all sub-processes of the engine are related to the selection and definition of time. The game loop and the time selection are closely related.

I.I Time

Time in the *Quake* engine is relative to real time. The client provides the time of an action performed, which is passed to the server along with the details of the event/message. The server then sends responses and details of events from other clients, or on the server itself, which are also in the context of real time.

On the client side, the time is obtained from *ct.time. Ct* is a struct of type *client_state_t*, which contains more information relative to the individual client.

I.II Game Loop

The following[2] is a summary of the game loop in *Quake*:

```
WinMain
{
    while (1)
    {
        newtime = Sys_DoubleTime ();
        time = newtime - oldtime;
        Host_Frame (time)
        {
            setjmp
            Sys_SendKeyEvents
            IN_Commands
            Cbuf_Execute

            /* Network */
            CL_ReadPackets
            CL_SendCmd

            /* Prediction//Collision */
            CL_SetUpPlayerPrediction(false)
            CL_PredictMove

            CL_SetUpPlayerPrediction(true)
            CL_EmitEntities

            /* Rendition */
            SCR_UpdateScreen
        }
        oldtime = newtime;
    }
}
```

The entry point, *WinMain*, exists in *sys_win.c*.

As the comments imply, the game loop can mostly be simplified to three sections: Network, Prediction/Collision, and Rendition (Rendering).

First, the client reads all messages/events from the server, and processes them as necessary.

Next, all in-game entities have their predicted moves executed. Prediction exists to solve latencies issues; in most cases, based on a players previous move (they moved forward 1 unit), they will continue to to move in that direction. Note that this does not apply to actually predicting the player's choices. Instead, it applies to the actual movement of a player entity.

Finally, after all entities' changes have been applied, the objects and models are rendered. *SCR_UpdateScreen* coordinates the rendering functionality of *Quake*, which will be explained further on.

II. Human Interface Devices (HIDs)

Human interface devices in *Quake* are limited to mouse, joystick, keyboard, and DirectInput devices.

The implementation of mouse, joystick, and keyboard device functionality will be covered in another section.

DirectInput is an API created by Microsoft[5] that processes data from joysticks or any other supported game controller. *Quake* provides specific support for joysticks, but not for any other controllers.

The general DirectInput controller's configuration data is represented by the MYDATA struct:

```
typedef struct MYDATA {
        LONG lX;        // X axis
        LONG lY;        // Y axis
        LONG lZ;        // Z axis
        BYTE bButtonA;
        BYTE bButtonB;
        BYTE bButtonC;
        BYTE bButtonD;
} MYDATA;
```

Quake uses *dinput.h*, a Microsoft Windows API file, for the implementation of DirectInput.

III. Resource Management

Quake resources make use of caching, but the rest of the resources are stored either on the hunk or a zone.

III.I Zones

Memory allocated using zones are generally only for small strings and structures.

<u>zone.c</u>

- Z_ClearZone()
- Z_Free()
- Z_Malloc()
- Z_CheckHeap()

III.II The Hunk

The hunk manages the entire memory block allocated to *Quake* by the system. All resources not allocated to zones are allocated and accessed on the hunk.

<u>zone.c</u>

- Hunk_Check()
- Hunk_AllocName()
- Hunk_Alloc()
- Hunk_HighAllocName()
- Hunk_TempAlloc()

III.III Caching

The cache is for short-term usage of resources. After allocating something to the cache, you can manually remove it, or it will be automatically freed when the cache is flushed or if the cache is full and a new resource needs the space that the current resource is occupying.

zone.c

- Cache_Move()
- Cache_FreeLow()/Cache_FreeHigh()
- Cache_TryAlloc()
- Cache_Flush()
- Cache_Free()
- Cache_Alloc()
- Memory_Init()

IV. 3D Rendering/Drawing

Quake uses OpenGL for the drawing of all graphics in the game. However, comprehensive rendering takes place in preparation of sending rendered models to the gl_ functions.

IV.I Rendering Overview

render.h is a header file containing prototypes of most rendering functions. The rendering files use most of the math functions to calculate final model positions. These functions are contained in files prefixed with r_:

- r_edge.c/r_edgea.s
- r_efrag.c
- r_light.c
- r_sky.c
- r_sprite.c
- r_draw.c/r_drawa.s
- r_local.h
- r_main.c
- r_misc.c
- r_part.c
- r_shared.h
- r_surf.c
- r_vars.c/r_varsa.s
- r_aclip.c/r_aclipa.s

Rendering mostly revolves around Alias models, covered in the Game Object Models section.

IV.II Video Drivers

vid.h is a header file containing prototypes of most video driver-related functions. For Win*Quake*, there are three relevant files, all prefixed with vid_:

- vid_null.c
- vid_dos.c
- vid_win.c

While vid_dos.c and vid_win.c are self-explanatory, vid_null.c is a file containing the empty video driver functions, with a commented purpose "to aid porting efforts."

vid_win.c is one of the largest files in the source of the *Quake* engine. This file directly interfaces with the Win32 video driver, and calls upon many Win32 API functions.

IV.III OpenGL

OpenGL handles the drawing of the previously rendered models and scenery.

glQuake.h contains variable definitions and prototypes of all functions to be implemented by gl_ prefixed files. These files are categorized based on which part of the graphics they focus on – light, models, mesh, etc. - and implement the required functions. The gl_ files only require the rendered Alias models and scenes.

- gl_draw.c
- gl_mesh.c
- gl_model.h/gl_model.c
- gl_refrag.c
- gl_rlight.c
- gl_rmain.c
- gl_rmisc.c
- gl_rsurf.c
- gl_screen.c
- gl_test.c
- gl_vidnt.c
- gl_warp.c/gl_warp_sin.h

V. Character Animation/Sprites

Animation in *Quake* occurs only with alias models. The data structure for an alias model contains a number of animation frames. Animation frames are made up of the following:

- minimum X, Y, & Z values for the vertices
- maximum X, Y, & Z values for the vertices
- the name of the animation frame
- array of vertices for the animation

The minimum and maximum coordinate values designate the bounding box for the alias model. No animation vertices in the array of vertices may extend beyond the bounding box.

Animation frame vertices are the key to determining the animation of the alias model. The vertices contain a packed 3D vertex and a vertex normal. A constant scaling factor exists to calculate the true vertex value when packed.

The animation effect occurs by moving the vertices of the triangles that make up the alias model, depending on the current animation frame. A loop of these animation frames creates the illusion of movement.

V.I Sprites

Sprites are covered in the Game Object Model section.

VI. Physics

Physics functions in *Quake* all exist in *sv_phys.c*. The sv_ prefix implies that it is a server file, meaning that all physics calculations occur on the server side after the client sends a message/event.

Some operations can be sectioned into related categories.

VI.I Push Functions

The push functions are called when one entity pushes another entity. *SB_PhysicsPusher* decides what type of push operation to perform, out of the following:

- SV_PushEntity() - Does not modify velocity
- SV_PushMove() - Does modify velocity
- SV_PushRotate()

VI.II Main Functions

These are functions that call upon other physics functions, depending on the result of the calculations carried out.

SV_Physics() is the main physics functions. In a map of the physics functions, it is at the top, calling upon all other functions.

SV_Physics_Step() is the function controlling movement up steps.

SV_Physics_Toss() controls “toss, bounce, and fly movement. When onground(sic), do nothing.”

SV_Physics_Noclip() is for any moving object that does not obey physics.

SV_Physics_Follow() is for one object that directly following another object.

VI.III Consistent Functions

These are functions that will always be called on an object for each loop.

SV_AddGravity() modifies velocity to account for gravity.

SV_RunThink() runs the thinking of the object before moving it.

SV_CheckVelocity() bounds the velocity at max velocity when necessary.

VI.IV Miscellaneous Functions

SV_CheckAllEnts() “see[s] if any solid entities are inside the final position.”

SV_impact() runs touch functions on two entities once they have touched.

SV_FlyMove() is the basic solid body movement clip that slides along multiple planes.

ClipVelocity() is for entities that will “slide off the impacting object.”

SV_CheckWater() and SV_WallFriction() are both self-explanatory.

Finally, SV_WalkMove() is only called by players. It is the default physics function when attempting to walk.

VII. Collision Detection

Quake makes heavy use of Binary Space Partitioning (BSP) for collision detection.

VII.I Binary Space Partitioning

This process is defined as a subdivision of a space into a combination of planes.

To check for a collision, you look at the sequence of planes of two entities that are decidedly close enough to maybe be colliding. Then, you compare the planes of one entity to the planes of the other, and look for a pair of planes across entities where, for one of the planes, one end is on the positive side of the second plane, while the other end is on the negative side of the second plane.

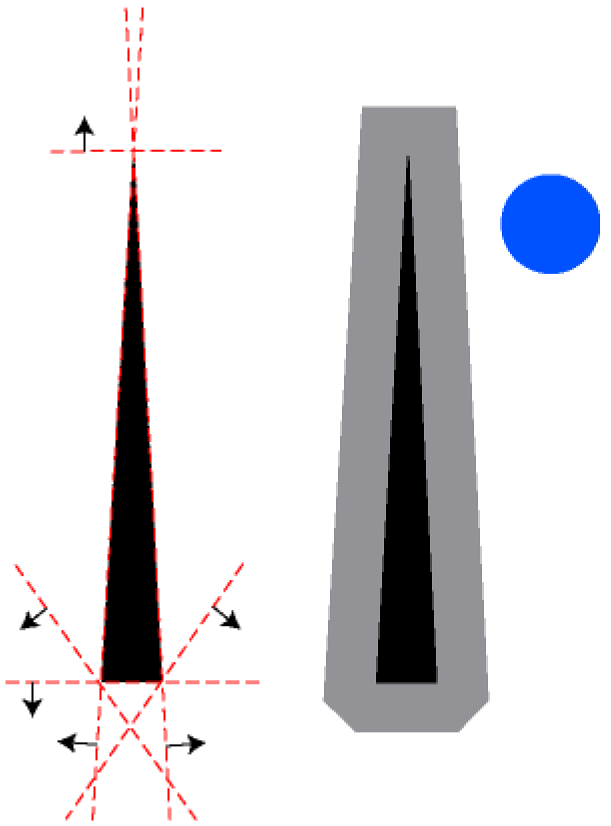

Figure 1. An example of the planes calculated via BSP for a triangle to be used in collision detection[3]

VII.II Implementation

In *world.c*, the function SV_FindTouchedLeafs() operates on two entities. It recursively determines if there is a collision between the two entities. Math functions such as Box_On_Plane_Side() are used in these calculations.

VIII. Rigid Body Dynamics

Quake minimally implements rigid body dynamics with its physics system.

A key component of this system is that it followed some equations of motions: usually Newton's laws of motion. The physics functions described in section IV have SV_AddGravity(), SV_CheckWater(), SV_WallFriction(), and SV_WalkMove() that all help to enforce a realistic physics system.

However, *Quake* does not support and rag-doll physics, or any other advanced rigid body dynamics systems.

The wireframe of alias models, which is explained in the next section, are what the physics functions rely on to operate.

IX. Game Object Models

There are three main game object models in *Quake*: aliases, models, and sprites.

IX.I Alias Model

Alias models represent players, objects, or monsters (all of which are often referred to as entities). They are the most frequently used game object model in *Quake*, since they can be animated.

The following is an excerpt from the *Quake* source of the definition of the alias model struct:

```
typedef struct
{ long id;              // 0x4F504449 = "IDPO"
                        // for IDPOLYGON
  long version;         // Version = 6
  vec3_t scale;         // Model scale factors.
  vec3_t origin;        // Model origin.
  scalar_t radius;      // Model bounding radius.
  vec3_t offsets;       // Eye position (useless?)
  long numskins ;       // number of skin textures
  long skinwidth;       // Width of skin texture
                        // must be multiple of 8
  long skinheight;      // Height of skin texture
                        // must be multiple of 8
  long numverts;      // Number of vertices
  long numtris;       // Number of triangles
                      // surfaces
  long numframes;     // Number of frames
  long synctype;      // 0= synchron, 1= random
  long flags;         // 0
  scalar_t size;      // average size of
                      // triangles
} mdl_t;
```

Alias models have a list of 3D vertices, which represent triangles in 3D space[7]. The triangles compose the wireframe of the alias model.

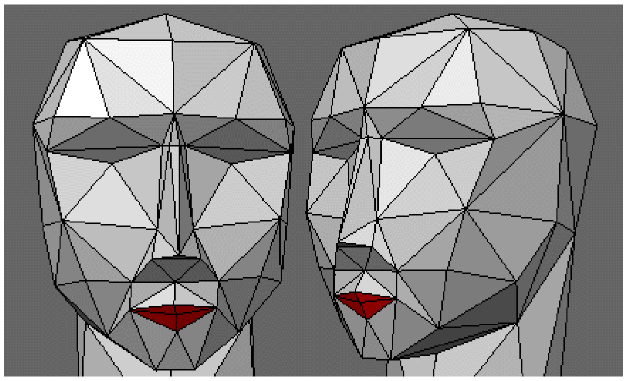

Figure 2. An example wireframe of an Alias model, composed of the triangles defined by a list of vertices

After the wireframe is composed, a texture is applied to the wireframe. Essentially, there is one texture and a list of 2D vertices. Each 2D vertex corresponds to a 3D vertex on the wireframe, to determine where to place a section of the texture onto the model.

The models also contain the animation frames used for a model.

IX.II BSP Model

A BSP Model is almost entirely made up of a struct defined as "Model". "The name Model refers here to either a big zone, the level, or smaller independent parts inside that zone, like the grid bars on level TEST1, that open with a push on the switch."[x]

```
typedef struct
{ boundbox_t bound;         // The bounding box of

                            // the Model
 vec3_t origin;     // origin of model,

                            // usually (0,0,0)
 long node_id0;     // index of first BSP node
 long node_id1;     // index of the first

                            // Clip node
 long node_id2;     // index of the second

                            // Clip node
 long node_id3;     // usually zero
 long numleafs;     // number of BSP leaves
 long face_id;              // index of Faces
 long face_num;     // number of Faces
} model_t;
```

IX.II Sprite Model

Sprites are generally reserved for very detailed, static objects. Compared to Alias models and BSP models, they render much faster. Some examples of sprite models used in *Quake* are explosions or barrels.

At a basic level, sprites are a list of 2D pictures organized into lumps. The following is the struct definition of a sprite:

```
typedef struct
{ char name[4];       // "IDSP"
  long ver1;          // Version = 1
  long type;          // See below
  float radius;       // Bounding Radius
  long maxwidth;      // Width of the

                      // largest frame
 long maxheight;      // Height of the

                      // largest frame
 long nframes;        // Number of frames
 float beamlength;
 long synchtype;      // 0=synchron 1=random
} spr_t;
```

and the following is the struct definition of a 2D picture of a sprite:

```
typedef struct
{ long ofsx;  // horizontal offset, in 3D space
 long ofsy;   // vertical offset, in 3D space
 long width;  // width of the picture
 long height; // height of the picture
 char Pixels[width*height];  // array of pixels
(flat bitmap)
} picture;
```

X. Events and Message Passing

There are two types of event and message passing in *Quake*: over the network (between client and server) and key events.

X.I Network Events

The following are functions used in message passing between server and client:

- NET_GetMessage()
- NET_SendMessage()
- NET_SendUnreliableMessage()
- NET_SendToAll()

The methodology behind sending and processing events and messages is similar in *Quake* to all other notable game engines:

1. The client performs an action;
2. The client sends a message containing the event details to the server;
3. The server receives the event, and processes it as necessary;
4. The server sends a message to clients containing new changes, and;
5. The client processes the message and the changes are displayed locally.

X.II Key Events

The mouse passes MouseEvents to the KeyEvent handler.

The keyboard passes KeyEvents straight to the KeyEvent handler. This handler checks the state of the game and the value of the KeyEvent, and calls other functions accordingly.

On Windows, TranslateMessage() and DispatchMessage(), two Win32 API functions, are sometimes used depending on the KeyEvent.

XI. Game Audio

The game audio functionality is split up into many files prefixed with snd_.

snd_dma.c is the main control for any streaming sound output device, DMA standing for Direct Memory Access.

snd_dos.c contains sound functions relating to interfacing with DOS.

snd_gus.c contains sound functions that interface with sound cards of the type GravisUltraSound (GUS).

snd_mem.c is used for sound caching.

snd_mix.c/snd_mixa.s is commented as "portable code to mix sounds for **snd_dma.c**"

snd_null.c is used at compile time when compiling a version of *Quake* with no sound.

Finally, **snd_win.c** contains sound functions for interfacing with Windows 95.

XII. Devices/Hardware

Mouse and keyboard input are the two main devices specifically covered by *Quake.*

XII.I Mouse

All mouse-related files are prefixed with in_. Mouse files are split up to interface with specific OSs.

in_dos.c handles DOS mouse controls.

in_null.c is a file with empty functions for systems with no mouse.

in_win.c contains Windows 95 mouse and joystick code.

Common to the DOS and Windows files are many functions. Initialization of the mouse, starting and stopping mouse control, and the hiding and showing of the mouse cursor. In_MouseEvent() is the handler for all MouseEvents.

XII.II Keyboard

All keyboard functionality is covered in **keys.h** and **keys.c**.

The files cover handling input from the keyboard, setting up key bindings, and initialization of keyboard constants.

The KeyEvent handler is the most comprehensive function in the file. The handler checks the current state of the game and whether or not the selected key of the KeyEvent is bound at all.

XIII. Networking

Quake separates network functionality into separate files that interface with all possible network protocols a client may be using for an internet connection.

XIII.I net.h

This file is commented as "*Quake*'s interface to the network layer. It is used to find *Quake* servers, get info about them, and connect to them. Once connected, the *Quake* game protocol is used."

net.h defines structs used in the other network files:

- qsocket_t: struct for a network socket
- net_landriver_t: driver for lan connections
- net_driver_t: driver for internet connections
- hostcache_t: contains information about the host of a game

Additionally, the file contains function definitions for the basic network functionality described in the previous comment.

XIII.II net_ files

Files prefixed with net_ contain all necessary information to interface with many network protocols.

net_bsd.c

Initalization of internet driver and LAN driver states/values

net_bw.h/net_bew.c
Relates to a Beame and Whiteside TCP/IP stack

net_comx.c
Relates to a COM port (probably connecting to a modem)

net_dgrm.h/net_dgrm.c
Handles Datagrams
"A self-contained, independent entity of data carrying sufficient information to be routed from the source to the destination computer without reliance on earlier exchanges between this source and destination computer and the transporting network."[1]

net_dos.c
Gives names of functions to DOS for each network protocol that may be used

net_ipx.h/net_ipx.c
Relates to Internetwork Packet Exchange (IPX) is the OSI-model Network layer protocol in the IPX/SPX protocol stack.

net_loop.h/net_loop.c
Relates to the Loopback protocol; when the request is sent out, it is immediately received locally and treated as if it came from somewhere else

net_main.c
As the name implies, the main network functions. Most take one socket when applicable. More than likely calls upon the specific protocols and acts as a wrapper for those, since the files with the protocols handle the specific details.

net_mp.h/net_mp.c
Relates to multipath routing (using multiple alternative paths through a network, for benefits such as...increased bandwidth (wikipedia))

net_none.c
Sets Loopback information

net_ser.h/net_ser.c
Relates to the serial protocol

net_udp.h/net_udp.c
Relates to the UDP (User Datagram Protocol)

net_vcr.h/net_vcr.c
Commented with "This is the playback portion of the VCR. It reads the file produced by the recorder and plays it back to the host. The recording contains everything necessary (events, timestamps, and data) to duplicate the game from the viewpoint of everything above the network layer."

net_win.c/net_wins.h/net_wins.c
Relates to Winsock (TCP/IP)

net_wipx.h/net_wipx.c
Relates to Winsock IPX (Internetwork Packet Exchange)

XIV. Scripting

Clients can create scripts in the *Quake* console, or create macro files externally to executed from the console.

XIV.I Scripts

Scripts can be created with the following syntax[4]:

```
/<variable_name> <new_value>

/set <variable_name> <value>

/unset <variable_name>

$<variable_name> [gets the value of
                     the variable]
```

These commands are received in **cmd.h** and added to a command buffer. Cbuf_Execute() calls Cmd_ExecuteString when the command has been parsed.

Creation of new commands are processed in Cmd_Alias_f and Cmd_AddCommand.

XIV.IIMacros

Macros can be created externally and saved with a .cfg extension. They are then run in the console using:

```
/exec <macro_name>.cfg
```

Macros are a series of commands/scripts in a row that are processed by the exec command and sent to **cmd.h** to be run.

XV. Math

Math functions implemented in *Quake* are used for three main purposes: rendering calculations, collision detection, and physics calculations. All functions are contained within **math.s**, **mathlib.h**, and **mathlib.c**.

XV.I math.s

This file contains assembly implementations of three functions in **mathlib.h/mathlib.c**: Invert24to16, TransformVector, and BoxOnPlaneSide. These functions are explained in the next subsection.

XV.II mathlib.h/mathlib.c

This file contains the rest of the math functions:

- **VectorMA()** - vector multiply + add
- **DotProduct()** - dot product of two vectors
- **VectorSubtract()** - subtraction of two vectors
- **VectorAdd()** - addition of two vectors
- **VectorCopy()** - copies values of an in vector to an out vector
- **VectorCompare()** - checks if two vectors are equal
- **Length()** - computes the length of a vector
- **CrossProduct()** - cross product of two vectors
- **VectorNormalize()** - normalizes a vector
- **VectorInverse()** - computes the inverse of a vector
- **VectorScale()** - scales a vector by a constant
- **Q_log2()** - returns the log base 2 of a number
- **R_ConcatRotations()** - combine two matrix rotations (3x3 matrix)
- **R_ConcatTransforms()** - combine two matrix transforms (3x3 matrix)
- **FloorDivMod()** - Returns mathematically correct (floor-based) quotient and remainder for numer and denom, both of which should contain no fractional part.
- **Invert24To16()** - Inverts an 8.24 value to a 16.16 value
- **GreatestCommonDivisor()** - calculates the GCD of two numbers
- **AngleVectors()** - Given three angles, gives the resulting forward, right, and up vectors relative to the direction the three angles give

- **BoxOnPlaneSide()** - If the box, given by the two vectors, has the side near the given plane fully in view return 1; not in view, return 2; partially in view, return 3
- **anglemod()** - computes the angle in the 0-360 range

References

[1] RFC 1594 FYI Q/A - http://tools.ietf.org/html/rfc1594

[2] Fabien Sanglard's *Quake* Engine Code Review – http://fabiensanglard.net/*Quake*Source/index.php

[3] Joel Gompert's Collision Detection Tutorial – http://www.joelgompert.com/collision/collision.html

[4] Ez*Quake*: Scripting http://ez*Quake*.sourceforge.net/docs/?scripting

[5] Win32 API Documentation: DirectInput - http://msdn.microsoft.com/en-us/library/windows/desktop/ee416842(v=vs.85).aspx

[6] GitHub – id Software: *Quake* Source – https://github.com/id-Software/*Quake*/

[7] Quake Engine Specification – http://www.gamers.org/dEngine/*Quake*/spec/