

PEQuake3: Particle enhanced Quake 3

Designing a scriptable particle system for Quake 3

Ron J.G. Otten
Guido J.W.M. Smeets

April 2, 2007

2IV00 Additional Component Computer Graphics 2006/2007
Department of Mathematics & Computer Science
Technische Universiteit Eindhoven

0496145 Ron Otten <r.j.g.otten@student.tue.nl>
0497737 Guido Smeets <g.j.w.m.smeets@student.tue.nl>

Contents

1	Introduction	3
2	System design	4
2.1	The Quake 3 architecture	4
2.2	Adding injectors and forces to the scene	5
2.3	Script loading and parsing	6
2.4	Updating the particle simulation	7
2.4.1	Spawning particles	7
2.4.2	Keeping track of particles	8
2.4.3	Accumulating forces on particles	10
2.4.4	The integration function	10
2.4.5	World geometry collision	11
2.5	Rendering the particles	12
2.5.1	Preparations for rendering	12
2.6	Dealing with multiple scenes	12
3	Implemented types of forces	15
3.1	Falloff	15
3.2	Directional force	15
3.3	Omnidirectional force	16
3.4	Viscous drag force	16
3.5	Centripetal force	16
3.6	Gravitational force	16
3.7	Attracting force	17
3.8	Spring force	17
4	Use cases	18
4.1	‘TestInjector’ and ‘TestForce’ commands	18
4.1.1	Performance measurement	18
4.2	Attaching particle injectors and forces to player models	19
4.2.1	The tag system	19
4.2.2	Our tested player model	20

5	Future work	22
A	Requirements	23
A.1	Functionality	23
A.2	Interaction	24
A.3	Presentation	24
B	Bibliography	25

1 Introduction

The Quake 3 engine is the 3D engine that lies at the heart of id Software's video game Quake III Arena. The engine and game logic were released under the GPL license on August 19 2005. In their original form only a very rudimentary system for handling what are called 'particles' is included in the game logic. It allows the creation of a 2D sprite, giving it a direction and a velocity and then letting it loose to expend its designated lifetime.

To fulfill the assignment for the 2IV00 "Additional Component Computer Graphics" course at TU/e, we have chosen to build and integrate a more feature rich particle system directly into the Quake 3 engine. We plan to make use of its existing architecture as much as possible while keeping our particle system modular, bolting its core on top of the Quake 3 engine's existing functionality. This will allow easy integration into existing source trees for actual real world usage.

2 System design

Our particle systems are built from two base components: *injectors*, which inject particles into the system and *forces*, which affect particles' acceleration.

Per software requirements FUNC1 and FUNC2 (See appendix A.) we need to be able to **add these injectors and forces** to the *'scene'* being built.

Per our software requirements INTR3 and INTR4 these injectors and forces are supposed to be configurable through a scripting language. This means we will need a **script parser** and a way to **load a script**.

Once we can do all this, we still need a way to **update the particle simulation** and to **render the particles**.

In addition, the Quake 3 engine itself and the way our particle systems will be used give rise to the following:

- **The particle system needs to be fast.**
A good frame rate is paramount for the genre of video games the engine was designed for.
- **The particle system needs to be resource conservative.**
When the Quake 3 engine starts up it reserves a fixed amount of memory up front, the so-called *'hunk'*. The engine already uses a large part of the hunk itself, mostly for caching assets. Furthermore, cached assets are non-disposable; meaning they heap up over time before they are 'flushed' by simply resetting the entire game. We don't want to end up taxing the existing system by too much, because it would trigger this event much sooner.
- **The particles' movements only needs to *look* correct, not actually *be* correct.**
The particle system will only serve as a graphical effect, not a scientific simulation. We can cut corners to speed things up.
- **No dynamic memory allocation.**
Dynamic memory allocation is expensive. Memory also fragments over time, causing dynamic allocation to be ever *more* expensive. Because we value speed highly, we avoid dynamic allocation as much as possible.

2.1 The Quake 3 architecture

A joke that a colleague of ours once encountered on the internet, goes: "To bake a pie, first one must create the universe." Or, applied to our situation; to understand the how and why of designing a particle system for the Quake 3 engine, first one needs (some) knowledge of how Quake 3 itself is structured.

On the top level, Quake 3 consists of 4 components:

- **The engine,**
which handles AI (Artificial Intelligence), peripheral input, networking, audio and video rendering and several other things. Internally it consists of three statically linked libraries; The opponent AI, the 3D renderer and ‘everything else’.
- **The server,**
which contains the physics and rules and logic of the game running atop the engine.
- **The client,**
which hands out rendering instructions to the engine, based on events received from the server.
- **The menu system,**
which contains both the main menu and the in-game menus.

The server, client and menu system are either dynamic link libraries, linking to the engine at run time, or they are virtual machine code that is either JIT compiled or interpreted by a special virtual machine that is part of the engine.

The renderer in the engine works on a frame-by-frame basis and is stateless, requiring every ‘entity’ that needs rendering on a particular frame, to be added to a list before the renderer is told to render it all into that one frame. Furthermore, to improve performance the renderer keeps in memory a cached version of all assets used by the engine; sound samples, 3D meshes, textures, etc. Thus, before an entity can be queued into a list for rendering, its asset has to be ‘registered’ once, so a ‘handle’ to the cached asset can be obtained.

Both the action of adding an entity to the current frame and the action of registering an asset are instructions delivered by the client component. This communication happens through a main entry point of the engine. This facility for ‘system calls’ uses a simple list of parameters, of which the first identifies what system call is being made.

The client module contains a set of facade functions, one for each possible system call, which in Quake jargon are called ‘trap calls’. We submit an example of two of these; the trap calls to register a 3D mesh and to add this mesh to the current frame as an entity.

```
qhandle_t trap_R_RegisterModel( const char *name ) {
    return syscall( CG_R_REGISTERMODEL, name );
}

void trap_R_AddRefEntityToScene( const refEntity_t *re ) {
    syscall( CG_R_ADDREFENTITYTOSCENE, re );
}
```

Listing 1: Example trap functions

2.2 Adding injectors and forces to the scene

As mentioned before, Quake 3 requires all entities that need rendering on the current frame, to be queued into a list before the frame is rendered. Because the renderer is stateless, this happens every frame. Because we do not want to break with existing architecture, we keep our injectors and forces stateless as well.

We duplicate the existing storage mechanisms for the rendering queue and in this way create ‘rendering’ queues for injectors and forces. In the same way that Quake 3 adds `refEntity_t` structures to the renderer, we add `refInjector_t` and `refForce_t` structures.

```

typedef struct {
    int         channelID; // Separates simulation of the particle system into
                        // separate channels, minimizing computation and
                        // keeping different segments of the system from
                        // affecting one another when required.

    qhandle_t   hInjector; // Handle to the injector script stored in the engine
    vec3_t      axis[3];    // Orientation; axis[0] == direction
    vec3_t      origin;
} refInjector_t;

typedef struct {
    int         channelID; // Separates simulation of the particle system into
                        // separate channels, minimizing computation and
                        // keeping different segments of the system from
                        // affecting one another when required.

    qhandle_t   hForce;    // Handle to the force script stored in the engine
    vec3_t      axis[3];    // Orientation; axis[0] == direction
    vec3_t      origin;
} refForce_t;

```

Listing 2: `refInjector_t` and `refForce_t` structures

We create similar trap calls to the existing one for `refEntity_t`, which will allow us to add our injectors and forces to the current frame from within the client code, fulfilling FUNC1 and FUNC2 from appendix A.1 and INTR1 and INTR2 from appendix A.2.

```

void trap_R_AddRefInjectorToScene( const refInjector_t *ri ) {
    syscall( CG_R_ADDREFINJECTORTOSCENE, ri );
}

void trap_R_AddRefForceToScene( const refForce_t *rf ) {
    syscall( CG_R_ADDREFFORCETOSCENE, rf );
}

```

Listing 3: Adding forces and injectors

2.3 Script loading and parsing

Quake 3’s methodology of registering assets comes into play here. A previously omitted fact is that assets can not only be binary files, like meshes or textures, but also script files. In fact; one of Quake 3’s central features is a special ‘*shader*’¹ language, that allows the scripting of dynamic, animated textures.

The existing script handling routines are easy to adapt to another scripting language to fit other programmers’ needs. Not only do they do a decent single-pass job of parsing the files, they also have very good associated storage routines. These give fast lookups of already loaded scripts through a hash table. Add to that the fact that you’ll be using a unified script loading/parsing/storing architecture that doesn’t break with the existing code base and the choice is easily made. Therefore we duplicate this approach used on the `.shader` files for our

¹Not to be confused with vertex and pixel shaders of modern graphics hardware.

own `.force` and `.injector` files, meant for forces and injectors respectively. Similar to the file `/renderer/tr_shader.c` you can find this adapted code in the files `/renderer/tr_force.c` and `/renderer/tr_injector.c`.

In similar fashion to the trap calls for registering assets, we create trap calls which will allow us to register both forces and injectors from the client code.

```
qhandle_t trap_R_RegisterInjector( const char *name ) {
    return syscall( CG_R_REGISTERINJECTOR, name );
}

qhandle_t trap_R_RegisterForce( const char *name ) {
    return syscall( CG_R_REGISTERFORCE, name );
}
```

Listing 4: Registering forces and injectors

A typical, complete example of client code using a particle injector can be found in listing 5, where we define a simple function to directly add a particular injector to the game world. The case for a force is analogous to this.

```
void PlaceInjectorInWorld(char *scriptName, vec3_t pos, vec3_t *axis) {
    refInjector_t inj;

    inj.channelId = ENTITYNUM_WORLD;
    inj.hInjector = trap_R_RegisterInjector(scriptName);

    VectorCopy(pos, inj.origin);
    VectorCopy(axis[0], inj.axis[0]);
    VectorCopy(axis[1], inj.axis[1]);
    VectorCopy(axis[2], inj.axis[2]);

    trap_R_AddInjectorToScene(&inj);
}
```

Listing 5: An example of registering and adding a particle injector

2.4 Updating the particle simulation

2.4.1 Spawning particles

We can add particle injectors and forces to our rendered scene, but for the moment there is still no particle to simulate. We first need the injectors to start spawning them, which is represented by requirement FUNC3. (See appendix A.)

We implement FUNC3 by retaining a countdown timer on each injector script. Once the script's given periodic spawning interval has passed, we let all injectors instanced off of this script spawn their particles. There are a few particulars with this that are slightly troublesome though, and we will highlight them here.

First; where do we keep track of this timer? As stated in section 2.1 the engine is stateless, which means that if we need to preserve the timer's value across frames, we need to break that statelessness.

Luckily an instance somewhat like this already exists; Aforementioned shader language (See section 2.3) makes use of a timer embedded in each shader script memory structure to properly

display animated effects in the texture. As we have no better alternative, we settle on a similar approach giving rise to the code in listing 6.

```
injector_t *script;

script = R_GetInjectorByHandle(injector->hInjector);

// Count down the spawn time, but only if we haven't counted
// down on this frame already
if (script->spawncountdown_modifiedtime != servertime) {
    script->spawncountdown -= timestep;
    if (script->spawncountdown <= 0) {
        script->spawncountdown = script->spawntime;
    }

    script->spawncountdown_modifiedtime = servertime;
}

// Time to spawn
if (script->spawncountdown == script->spawntime) {
    ...
    ...
    ...
}
```

Listing 6: The particle spawning timer

As the observant can already see; in the code in this listing we've already taken care of the second issue as well. Namely; the code to update the injectors iterates over all injector *instances* in the scene, rather than over all injector *scripts*. To prevent modifying the timer multiple times within the same frame, along with the interval value itself we keep track of the last time that value was modified. If the current time (`servertime`) matches the last modified time, the timer was already modified.

2.4.2 Keeping track of particles

Now that the injectors can spawn particles, there is another issue to deal with, also related to the renderer's statelessness. When a particle has spawned it has to be retained over time, until it expires. (See `FUNC5` in appendix A.1.) Until the time it expires, it is then subject to the forces present in the simulation.

How do we keep track of these particles' states? It is certainly not an option to communicate the entire collection of particle states between the engine and the client component each time the client issues an order to update the particle simulation. This would incur twice the memory overhead: Once for storing the data inside the client component and once for keeping a temporary copy that the engine processes.

Again we find that we have to break the renderer's statelessness, but this time we don't have the luck of being able to add some fields to a pre-existing storage structure, as was the case in section 2.4.1. Instead we opted to create a data structure Quake 3 uses in several other places. We shall refer to it as a doubly & circularly linked list: A doubly linked list with the head element's `previous` pointer field pointing to the current tail element and the current tail element's `next` pointer field pointing to the head element.

At first sight the usage of a linked list seems to not comply with our wish to avoid dynamic memory allocation, but this is not true. In fact; it is perfectly possible to create a linked list using static allocation, as long as we define an upper bound on the number of elements in the list. As it makes no sense to overload a real-time renderer with too many particles to simulate and render anyway, this solves two issues at once; We have static allocation and we bound the number of particles to a reasonable amount.

This statically allocated, doubly & circularly linked list works by statically pre-allocating the maximum amount of particles, in the form of a simple one-dimensional array. Each element is linked to the next over its `next` pointer field. Next to this array, we maintain a *'free list'* pointer, which will point to the first free element of the array, and a separate list element which will serve as the head of the *'active list'*.

Now it simply becomes a matter of correctly updating the `next` and `previous` pointers correctly each time an element is made active or is freed. In this way we can even guarantee that the `previous` element of the active list's head is always the oldest active particle. A nice property which we can exploit when we are trying to 'allocate' more particles than the list has the capacity for; Should this happen we simply 'free' this `previous` element of the head, since it has had the most time in the system already.

See listing 7 for the completed code.

```

static particle_t    particles[MAX_PARTICLES];
static particle_t    *particles_free;
static particle_t    particles_inuse;

static void R_FreeParticle( particle_t* particle ) {
    if ( !particle->prev ) {
        Com_Error( ERR_DROP, "R_FreeParticle: not active" );
    }

    // remove from the doubly linked active list
    particle->prev->next = particle->next;
    particle->next->prev = particle->prev;

    // link back at the head of the free list
    particle->next = particles_free;
    particles_free = particle;
}

static particle_t* R_SpawnParticle() {
    particle_t *particle;

    if ( !particles_free ) {
        // No free particles, so free the one at the end of the chain,
        // removing the oldest active particle
        R_FreeParticle( particles_inuse.prev );
    }

    particle = particles_free;
    particles_free = particles_free->next;

    memset( particle, 0, sizeof( particle_t ) );

    // link into the active list
    particle->next = particles_inuse.next;
    particle->prev = &particles_inuse;
    particles_inuse.next->prev = particle;
    particles_inuse.next = particle;
}

```

```

    return particle;
}

void R_InitParticles( void ) {
    int    i;

    Com_Memset( particles, 0, sizeof( particles ) );
    particles_inuse.next = &particles_inuse;
    particles_inuse.prev = &particles_inuse;
    particles_free = particles;

    // singly link the free list
    for ( i = 0 ; i < MAX_PARTICLES - 1 ; i++ ) {
        particles[i].next = &particles[i+1];
    }
}

```

Listing 7: Particle storage through statically allocated, doubly & circularly linked lists

2.4.3 Accumulating forces on particles

Before we can run any integration function (See section 2.4.4.), we first need to know the cumulative force of all the particles. This means we need to accumulate all individual forces affecting a particle. We perform this by simple looping over all the forces in the scene. We check what type each force is and offload its handling to a function specifically designed for that type of force. Section 3 gives in depth information on the types of forces we have built.

Looping through each particle and performing an inner loop through the forces associated with that particle may seem more natural, because we also have to integrate the particle after accumulation. However, a few of the types of forces we wanted to build compute their effect on a particle by using related particles. It turns out that the implemented functions to handle those kind of forces become much less obfuscated when they can simply be passed the head of the particle list and loop through that list themselves.

2.4.4 The integration function

To actually update the particles' positions we need a proper integration function. By FUNC4 (See appendix A.1.) it needs to be as fast as possible, facilitating real-time computation. While as stated before; we don't require absolute scientific correctness, we do require the particles to look like they are behaving correctly. For a smooth movement, this means we still require a fairly high order accuracy.

Jonathan Dummer presents a nice integration scheme that fits these needs in his article "A Simple Time-Corrected Verlet Integration Method" ["lonesock" Dummer(2005)].

According to Dummer, under the right conditions Verlet integration is 4th order accurate. In contrast; The well-known Euler integration is only 1st order accurate and Runge-Kutta is only 2nd order accurate. Additionally, it is almost as easy (and fast) to compute as Euler integration.

While this sounds very promising, there is still a major downside to Verlet integration. Namely; it does not handle varying time steps well. That is a big problem for a real-time 3D

engine that by its nature has a varying frame rate and thus time step. There are some other issues with Verlet integration, but Euler integration has these as well. Empirically it suffers even worse under them, actually.

Dummer goes on to introduce Time-corrected Verlet integration, a version of the algorithm which solves the issues with varying time steps. He shows empirical evidence that this corrected algorithm indeed performs better than Euler integration, even on varying time steps.

The Time-corrected Verlet integration scheme works according to

$$x_{i+1} = x_i + (x_i - x_{i-1}) * \frac{\Delta t_i}{\Delta t_{i-1}} + a * \Delta t_i^2$$

, where x_i is the particle's position at simulation step i and Δt_i is the amount of time passed between simulation step $i - 1$ and i .

Note that the velocity term doesn't need to be retained in a separate field. Rather, it can be constructed from the position at step i and at the previous step $i - 1$. Also, the fraction of Δt terms only needs to be computed once per step, eliminating much of the equation's calculation time.

For how this actual formula is reached, we refer to the original article. The integration function, combined with the preceding accumulation of forces implements FUNC8 from appendix A.1.

2.4.5 World geometry collision

After particles' have had their current position updated by the integration function, we still have to take care to make sure they haven't passed into or through any solid objects. Having collisions between particles take place (FUNC15) would be too expensive, but it is certainly possible to have collisions between particles and the world geometry (FUNC13) and between particles and dynamic world entities (FUNC14).

To implement collision with the world geometry, we really only need 3 things: The old position of the particle, the new position of the particle and a way to trace a ray inside the game world, between these two positions. Luckily, the Quake 3 engine provides just such a thing; The `CM_BoxTrace` function.

```
void CM_BoxTrace ( trace_t *results, const vec3_t start, const vec3_t end,
                  vec3_t mins, vec3_t maxs, clipHandle_t model, int brushmask, int capsule );
```

Listing 8: The `CM_BoxTrace` function

A call to `CM_BoxTrace` returns data of the trace into the `trace_t` data structure, which we can use to judge whether or not our particle has impacted on a surface.

If it has, then we can use that same `trace_t` structure to retrieve the normal of the plane on which we impacted, leading to an easy way to not only detect collision, but also to bounce a particle off of the world geometry.

$$\vec{d}_{new} = \vec{d}_{old} - 2(\vec{d}_{old} \cdot \vec{n})\vec{n}$$

, where \vec{d} is the particle's direction and \vec{n} the surface normal.

If called with different parameters, we can also use `CM_BoxTrace` to perform collisions against dynamic world entities, but we won't. It adds additional computational overhead for an effect that won't look convincing anyway, as traces against dynamic world entities are not done by their mesh surface, but by a rectangular bounding box.

2.5 Rendering the particles

Quake 3's renderer is subdivided into two sections; a front end, into which the programmer puts the objects that need to be rendered, and a back end which takes all the data that the programmer has put into the front end's buffer and renders it.

To render our particles we've created a routine that converts the particle into a correctly oriented billboard sprite, and then uses existing methods to feed this oriented quadrangle to the renderer. This implements requirement PRES1. (See appendix A.3.) We would have liked to also implement PRES2, and give the option to use 3D meshes as particles, but constructing the associated rendering function would be too much time better invested in other parts of the project.

We acknowledge that building the rendering function like this *does* leave a few issues which future work will have to resolve. All of these issues are related to Quake 3's '*portal*' mechanism, which is used to produce mirrored surfaces. The only way to solve these problems is to construct a completely new rendering type and back end rendering code specifically for particles. Regrettably this was too much additional work to complete within the project's time frame.

2.5.1 Preparations for rendering

Before we can convert the particle into a billboard sprite we need to know a few things; The texture to use on the sprite, the sprite's size, its coloring and its opacity.

Of these scripted options, the size, coloring and opacity are all specified through a start and end value, between which we interpolate linearly. (See PRES3 and PRES4 from appendix A.3.) This can give us nice smoothly appearing and disappearing particles for use in things such as smoke screens.

2.6 Dealing with multiple scenes

In the above, we have laid out a complete system for managing a scripted particle system. However, it is all based on the assumption that there is only a single active scene being rendered each frame. Unfortunately, the Quake 3 engine doesn't exactly work in that fashion. Rather; it is possible to successively render multiple scenes into the same frame, such as in listing 9. Moreover; the number of possible scenes is not fixed and may even vary over time.

```

refdef_t      refdef;
refEntity_t   ent;

...

// Draw the main 3D view as it is at this point
trap_R_RenderScene( &cg.refdef );

// Initialize a new (overlay) scene
trap_R_ClearScene();

// Put something into the overlay scene
trap_R_AddRefEntityToScene( &ent );

// Render the overlay scene on top of the 3D view
trap_R_RenderScene( &refdef );

```

Listing 9: An example of rendering multiple scenes

Particles belong to the scene in which they are spawned by an injector and must only be put into the simulation for that specific scene. Because the number of scenes is not fixed, we can't simply create a different list of particles for each scene. We could, if we would allow dynamic allocation of memory, but as mentioned before; that's far too expensive, considering the number of scenes can change every rendered frame.

Instead we maintain what we already have and simply add an additional field of information to the particle state. Here we store an ID value that matches with the scene the particle belongs to. We then modify our trap call to update the particle simulation to include this scene ID. This leads to the relatively small burden on the programmer of the client logic to keep track of the proper ID to use, but that is a necessary evil.²

Because we now have multiple scenes to deal with (and only part of those scenes may take place in the actual game world), we also need a way to tell the particle simulation to include the game world into the simulation update, or to leave it out. This is important for the correct functioning of collisions with the world geometry. (See section ??.) We can solve this by simply adding a boolean parameter to the update trap call, which turns the world on or off.

Listing 10 contains an updated version of the example from listing 9. It has the (modified) instructions for particle systems in multiple scenes added to it.

```

refdef_t      refdef;
refEntity_t   ent;
refInjector_t inj;

...

// Draw the main 3D view as it is (with Scene ID=0 and the game
// world turned on) at this point
trap_R_UpdateParticlesInScene(0, qtrue, cg.frametime, cg.time);
trap_R_RenderScene( &cg.refdef );

// Initialize a new (overlay) scene
trap_R_ClearScene();

```

²This may be alleviated somewhat by defining all the used scene ID numbers in an enumeration datatype. The Quake 3 client logic has several instances of this being done.

```
// Put something into the overlay scene
trap_R_AddRefEntityToScene( &ent );
trap_R_AddRefInjectorToScene( &inj );

// Render the overlay scene (with Scene ID=1 and the
// game world turned off) on top of the 3D view
trap_R_UpdateParticlesInScene(1, qfalse, cg.frameTime, cg.time);
trap_R_RenderScene( &refdef );
```

Listing 10: An example of rendering multiple scenes with particle systems

3 Implemented types of forces

A particle isn't that interesting to look at until it starts reacting to forces. We've defined a solid set of forces that should provide enough different ways to affect a particle to get a lot of interesting looking effects.

We provide directional, omnidirectional, drag, orbital and gravitational forces, which all affect individual particles, and attraction and spring forces, which all affect pairs of particles. Naturally, the latter two are more expensive. With n particles, the former set is actually of order $O(n)$ complexity, while the latter is of order $O(n^2)$ complexity.

3.1 Falloff

Before we treat each different type of force, we first discuss the notion of falloff: When a particle gets further away from the source of the force, the affect of the force on the particle diminishes.

In real world situations the falloff would generally be proportional to the inverse of the squared distance between the particle, and the force's source.³ In our simulation instead we opted to use a linear falloff based on a scriptable value, because doing so gives a content author or programmer that much more control over the forces (s)he wants to use.

The falloff between points \bar{a} and \bar{b} with falloff script value r would be calculated as follows.

$$f = 1 - \frac{\|\bar{a} - \bar{b}\|}{r}$$

3.2 Directional force

A directional force is the most basic of forces and can be used when particles should be pushed in a certain direction. Wind is an example of such a force.

The formula to calculate the directional force component is as follows.

$$\vec{F}_{dir} = Mf\vec{d}$$

, where M is the force's magnitude as defined through the scripting interface, f is the falloff ratio as discussed in section 3.1, and \vec{d} is the direction of the force.

³When dealing with forces that operate on pairs of particles forces, the force's source would be the second particle in the pair.

3.3 Omnidirectional force

The omnidirectional force pushes particles away from the point of origin of the force. It can be used well when simulating all kinds of explosion effects.

The formula is slightly more complex than that of the regular directional force, but only because the direction of the force has to be calculated out of two parts of position information.

$$\vec{F}_{omni} = Mf \frac{\bar{p} - \bar{s}}{\|\bar{p} - \bar{s}\|}$$

, where M is the force's magnitude, f is the falloff ratio, p is the particle's position and s is the force's position.

3.4 Viscous drag force

Viscous drag is a force that acts to resist motion, and is only affected by the particles velocity.

$$\vec{F}_{drag} = M(\bar{p}_{new} - \bar{p}_{old})$$

, where M is the force's magnitude, the so-called drag coefficient [Bourke(1998)], \bar{p}_{new} is the particle's current position and \bar{p}_{old} is the particle's position on the previous frame.

3.5 Centripetal force

We do not actually have a full implementation for centripetal (and centrifugal) forces, but we note here that with proper tweaking of script parameters, a similar effect can be achieved with an omnidirectional force that is set to a negative magnitude.

3.6 Gravitational force

We call this gravitational force, though it isn't really a force. It has just been made configurable *as* a force, to provide programmers and content authors the option to enable or disable gravity on their particles.

It is implemented as a special case of the directional force. The gravitational force always has a downward direction, and its magnitude is affected by the particles mass as follows.

$$\vec{F}_{grav} = m \vec{d}$$

, where m is the particle's mass, and \vec{d} is the downward vector of unit length.

We multiply the downward vector by the particle's mass, because in the integration step (See section 2.4.4.) the accumulated force (of which this force would then be part) is divided by m to obtain acceleration a . In this way we assure that gravity will contribute a correct value to that a .

3.7 Attracting force

The attracting force enables particles to perform a gravitational pull on each other. The force between particles p_1 and p_2 is calculated as follows.

$$\vec{F}_{attract} = GfMm_1m_2$$

where G is the in-game equivalent of the universal gravitational constant, f is the falloff based on the distance between both particles, and m_1 and m_2 are the mass of particle p_1 and p_2 respectively. To give programmers and content authors more control, we've allowed the influence of this force to be tweaked by changing scripted magnitude M .

3.8 Spring force

Our spring force is based on Hooke's Spring Law. The idea behind this force is to let the particles find an equilibrium in which they are all a certain distance; the radius, apart from each other. If two particles are closer than the specified radius, they will repulse each other, whereas if they are further apart than the specified radius, they will attract each other. After a while their relative positions stabilize, and an equilibrium is found.

The force between two particles a and b is calculated as follows [Bourke(1998)]:

$$\vec{F}_{spring} = - \left[k_s(\|x_a - x_b\| - r) + k_d(v_a - v_b) \frac{x_a - x_b}{\|x_a - x_b\|} \right] \frac{x_a - x_b}{\|x_a - x_b\|}$$

, where k_s is known as the spring constant, k_d is the damping constant, r is the radius at which equilibrium is reached and x and v are the usual notations that stand for position and velocity. The spring constant k_s can be changed through the scripting interface as the scripted force's magnitude.

4 Use cases

We now have a completely working scriptable particle system, but of course this means nothing if we do not also have a way of *using* said particle system. Therefore we have implemented two use cases to illustrate possible use of the system.

4.1 ‘TestInjector’ and ‘TestForce’ commands

One useful set of commands to have around is always that of the “Quickly test this!” variety. With facilities already present in Quake 3’s default game logic (in the client module), we’ve built a pair of commands to be input on Quake 3’s ‘*console*’; a command line facility reachable through the tilde key.

The `TestInjector` and `TestForce` commands take as a parameter the name of an injector or force script respectively. They then use some of the game logic’s existing structures to instance said injector or force script a short distance in front of the player, where it will remain for the duration of ten seconds.

Using these test commands we ourselves have been able to quickly debug several scripts, including one used on our next use case.

Some users might even find it a fun test bed, to experiment with several properties of the injectors and forces.

4.1.1 Performance measurement

We have used the test commands crated in this use case to measure performance of our particle system as the number of particles and forces offset against the frame rate in frames per second.

Interestingly, on our test machine it shows that the particle simulation doesn’t have a real impact on the frame rate under regular use. We’ve had to seriously tax the system with over 200 particles, each having inter-related spring forces (meaning we had 40000 or more active forces), before we noted a serious drop in the framerate.

Number of particles	Number of regular forces	Number of inter-particle forces	framerate (frames/sec)
0	0	0	90+
10-20	3	0	88-90
10-20	0	1	88-90
50-60	3	0	83-86
50-60	0	1	83-86
100-110	3	0	78-81
100-110	0	1	77-80
200+	3	0	75-80
200+	0	1	62-64
200+	0	2	50-53

4.2 Attaching particle injectors and forces to player models

The communities revolving around the games built on the Quake 3 engine have always had a steady supply of extra, community-built content.

Not only ‘maps’; game worlds to play in, but also ‘player models’; the 3D animated meshes that serve as the player’s avatar in the game world, have always been popular to create yourself.

For that reason we felt we should open up our particle system to the people creating this content as well. Hence our second use case; the ability to couple injectors and forces to player models.

4.2.1 The tag system

To actually couple an injector or force to such a player model, we need to give the artist a method to provide a point of reference: *Where* on the player model does (s)he want to place the injector or force?

Luckily for us; Quake 3 has a native mechanism for this, based on something called a ‘tag’. A tag is nothing more than a special kind of triangle an artist can put into his or her mesh structure. When the artist then exports the work from his or her favored 3D rendering suite, the available conversion tools will pick up on this triangle and mark it. From then on, special trap calls to the Quake 3 engine can be used to track exactly the position and orientation of the tag on a current animation frame of the mesh.

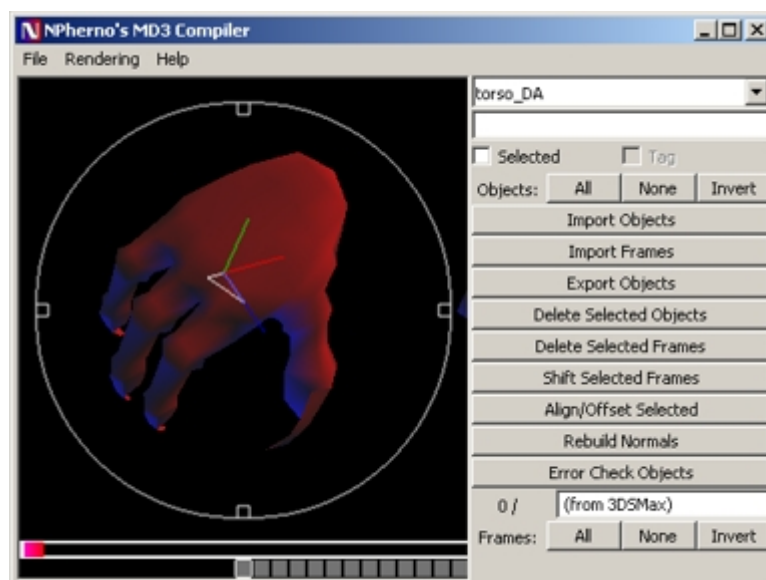


Figure 1: A tag in a mesh, as displayed by the tool “Npherno’s MD3 Compiler”

Artists working with the Quake 3 engine should already be comfortable with the concept of tags, as it is mandatory to split a player model’s mesh into three parts (legs, torso and head)

and use tags as places of reference to how those three parts should connect together again. This is a feature used to mix and match animations in the upper body and lower body (such as simultaneously running and firing a weapon).

With a small bit of script parsing and other pieces of code added to the game logic to correctly spawn the injectors and forces indicated by the script, all the artist has to do is provide said coupling script.

As a reference listing 11 contains the coupling script for our own test model.

```
// tagname, bodypart, force/injector, scriptname, active on death (yes/no = 1/0)

// MAIN BODY
tag_torso legs force DarkArchon_BodyOrbPull 0
tag_torso legs force DarkArchon_BodyOrbPush 0
tag_torso legs injector DarkArchon_BodyOrb 0

tag_head torso force DarkArchon_BodyOrbPull 0
tag_head torso force DarkArchon_BodyOrbPush 0
tag_head torso injector DarkArchon_BodyOrb 0

// HANDS
tag_handL torso injector DarkArchon_Hands 0
tag_handR torso injector DarkArchon_Hands 0
tag_handL torso force DarkArchon_HandsPush 0
tag_handR torso force DarkArchon_HandsPush 0

// WEAPON
tag_weapon torso injector DarkArchon_Hands 0
tag_weapon torso force DarkArchon_HandsPush 0

// LOWER BODY CLOUD SPRAY
tag_torso legs injector DarkArchon_Cloud 0
tag_torso legs force DarkArchon_CloudGravity 0
tag_torso legs force DarkArchon_CloudSpread 0
```

Listing 11: A typical coupling script, as used by our test model

4.2.2 Our tested player model

Finally, we wanted to give a clear indication of the usefulness of being able to attach injectors and forces to player models. For that, we needed a type of player model that without this kind of functionality could not have been created properly.

As it is *also* a popular activity for model artists to create 3D representations of their favorite characters from television series, cartoons or even other video games, we settled on something similar: The Dark Archon character from Blizzard Entertainment’s successful real-time strategy game StarCraft.

As you can see in figure 2 this character is a ghostly appearance suspended in some kind of field of energy. That field is exactly where we would require particles to make the model look good.

Our test model⁴ of this character can be seen in figure 3 in a side by side comparison of the

⁴Model’s texture courtesy of P.M.A. Otten

model *without* the particle field and *with* the particle field. Quite a difference! And all it took were a few particle injectors, omnidirectional forces and an attracting force.

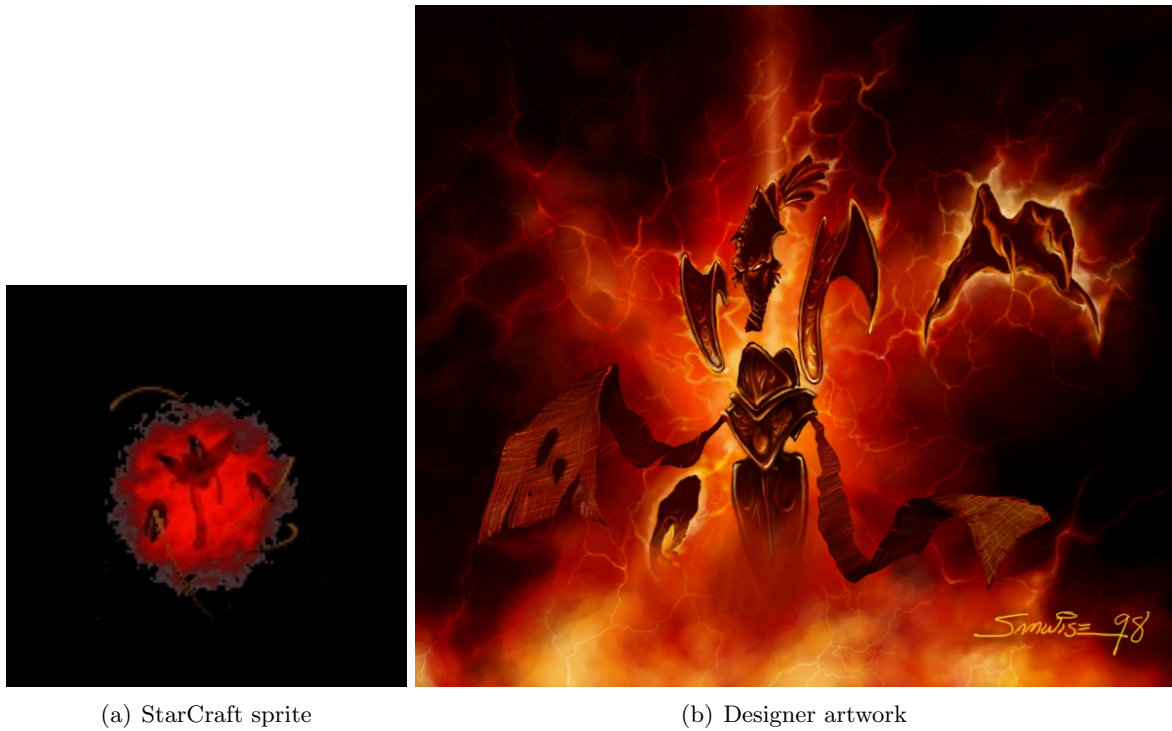


Figure 2: Dark Archon references



Figure 3: Our Dark Archon test model

5 Future work

While we have been able to implement the core requirements we established in our preliminary report, there are still a few things missing we would like to complete or see completed in the future. Requirement INTR6, for instance, which specifies the ability to place injectors and forces directly on the world environment when building it in a world editor.

There is also still the matter of the particle rendering function not behaving entirely as it should, which requires writing some very technical rendering code.

We've also garnered the interest of a few people working on other open source projects stemming from Quake 3 and we've already enthusiastically offered to make our project public.

After fully completing our implementation we may try to get the project integrated into one of the larger running source trees, such as ioQuake3 (<http://ioquake3.org/>), as well.

Appendices

A Requirements

The following appendix contains the software requirements as established during the preliminary report that required evaluation before this project would be accepted for the 2IV00 “Additional Component Computer Graphics” course.

A.1 Functionality

ID	Description	Priority
FUNC1	particle injectors can be added to the currently rendering frame	M
FUNC2	forces can be added to the currently rendering frame	M
FUNC3	particle injectors generate particles at configurable intervals	M
FUNC4	calculate in ‘real-time’ the updated particle positions for the currently rendering frame	M
FUNC5	particles expire when a maximum lifetime has been met	M
FUNC6	particles have position, velocity and acceleration	M
FUNC7	particles have angular speeds	C
FUNC8	particles are affected by forces	M
FUNC9	particles can be given a maximum distance constraint	M
FUNC10	particles can be given a minimum distance constraint	M
FUNC11	particle injectors have a ‘channel’, to filter which forces affect their generated particles	M
FUNC12	forces have a ‘channel’, to filter which particle injectors’ generated particles they affect	M
FUNC13	particles can collide with the map/world geometry	M
FUNC14	particles can collide with dynamic game entities	S
FUNC15	particles can collide with particles	C
FUNC16	particles can spawn new particles at the end of their lifetime	W

A.2 Interaction

ID	Description	Priority
INTR1	mod authors can pass a configured <code>refInjector_t</code> memory structure to the engine to fulfill FUNC1	M
INTR2	mod authors can pass a configured <code>refForce_t</code> memory structure to the engine to fulfill FUNC2	M
INTR3	content authors can write a '.injector' scriptfile to configure a particle injector	M
INTR4	content authors can write a '.force' scriptfile to configure a force	M
INTR5	content authors can attach scriptfiles from INTR3 and INTR4 to a player model 'tag'.	S
INTR6	content authors can attach scriptfiles from INTR3 and INTR4 to entities on a map	C

A.3 Presentation

ID	Description	Priority
PRES1	particles can be presented as 2D sprites during play	M
PRES2	particles can be presented as 3D meshes during play	C
PRES3	particles can be scaled with lifetime	M
PRES4	particles can be faded with lifetime	M

B Bibliography

- [Bourke(1998)] Paul Bourke. Particle system example, Feb 1998. URL http://ozviz.wasp.uwa.edu.au/~pbourke/modelling_rendering/particle/.
- [“lonesock” Dummer(2005)] Jonathan “lonesock” Dummer. A simple time-corrected verlet integration method. GameDev.net, Feb 2005. URL <http://www.gamedev.net/reference/articles/article2200.asp>. Date of publication on GameDev.net may not necessarily correspond to date the article was written.