# CERN Destroys Alien Ship

Muhamed Razalee Yusoff, Neil Edelman, Daniel Smilowitz, Olivier Shelbaya

2009-04-02

## 1 Introduction

Antennae are devices that have become commonplace in today's world. They allow an analogue (or digital) signal carried via wire to be transmitted through space by emitting electric fields, which then induce a current in another antenna, further away. As a consequence of the varying electric field, a magnetic field is simultaneously generated, although it is smaller than the electric field by a factor $\frac{1}{c}$. For high powered antennae, as employed in this simulation, the magnetic field becomes non-negligable.
This becomes very important in the practical concept demonstrated in this work: the destruction of a spaceship. Spaceships are generally terrifying vessels of destruction which can allow for the birth of interplanetary empires of evil. It is therefore very important to know how to destroy them, theoretically anyways...

A high powered EM wave (Gigahertz range frequency) is sent via high gain antenna (i.e. most of the signal entering the antenna by wire exits as an EM field) into an unsuspecting spaceship, driving the spaceship to its **resonance frequency**, causing the vessel to explode. Spaceships *do* present many similar points to oscillators. For one, they have matter, and matter oscillates. Our hypothesis is therefore verified, and the math may begin.

## 2 Theory

In order to accomplish simulation, the following concepts were employed:
The poynting vector is derived from the definition of **energy** in an electromagnetic field:

$$U_{em} = \frac{1}{2}\int\left(\epsilon_0 E^2 + \frac{1}{\mu_0}B^2\right)d\tau \qquad (1)$$

Using the above definition, along with the lorentz force law and employing vector product rules, it is possible to find that:

$$\frac{dW}{dt} \propto -\frac{1}{\mu_0}\oint(\mathbf{E}\times\mathbf{B})\cdot d\mathbf{a} \qquad (2)$$

That is, the energy per unit time and area can be described by the **Poynting Vector**:

$$\mathbf{S} = \frac{1}{\mu_0}(\mathbf{E}\times\mathbf{B}) \qquad (3)$$

As can be seen by analyzing the cross product of the definition, since E and B are perpendicular by definition, the poynting vector points in the direction of propagation of the EM wave.

The next definition of interest is that of the **energy and momentum** for an EM wave. Although the physical quantity is small, electromagnetic waves nevertheless carry a small quantity of energy (obviously much less than a sound wave, for example). Equation (1) describes the energy per unit volume (in 3-space) of an EM wave.
The **momentum** of an electromagnetic wave is, by definition, in terms of the speed of light, $c$

$$\mathbf{P}_{em} = \frac{1}{c^2}\mathbf{S} \qquad (4)$$

Since we are in the presence of time-dependant potentials and fields, in order to calculate the fields at a distant point of interest as a function of time, **Jefimenko's Equations**, which are the electrodynamic generalisations of the coulomb and biot-savart laws were employed.

The electric field Jefimenko equation is:

$$\boldsymbol{E}(\boldsymbol{r},t) = \frac{1}{4\pi\epsilon_0}\int\left(\frac{\rho(\boldsymbol{r'},t_r)\,\boldsymbol{R}}{R^3} + \frac{\boldsymbol{R}}{R^2c}\frac{\partial\rho(\boldsymbol{r'},t_r)}{\partial t} - \frac{1}{Rc^2}\frac{\partial\boldsymbol{J}(\boldsymbol{r'},t_r)}{\partial t}\right)d\tau' \qquad (5)$$

While the Magnetic field Jefimenko equation is:

$$\boldsymbol{B}(\boldsymbol{r},t) = \frac{\mu_0}{4\pi}\int\left(\frac{\boldsymbol{J}(\boldsymbol{r'},t_r)}{R^2} + \frac{\dot{\boldsymbol{J}}(\boldsymbol{r'},t_r)}{cR}\right)\times\hat{R}\ d\tau' \qquad (6)$$

Taking:

$$\boldsymbol{J}(r',t') = \boldsymbol{J}(\boldsymbol{r'})e^{i\omega t}e^{-i\frac{\omega}{c}R} \qquad (7)$$

We approximate:

$$e^{i\omega t} = cos\left(\frac{\omega}{c}R\right) + isin\left(\frac{\omega}{c}R\right) \qquad (8)$$

Finally, since all of the aformentionned quantites pertain to phenomenon which are oscillatory by construction, their instantaneous values are of little use for modelization. The **time average** is used, as it gives a better representation of the average value, for one cycle. The time average operator is defined as:

$$\langle\rangle = \frac{1}{T}\int dt \tag{9}$$

where T is the period of oscillation, in seconds.

## 3 Results

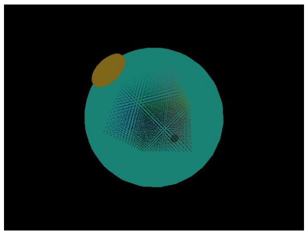

Figure 1: Step 1 : Alien Spaceship spotted entering Earth's orbit

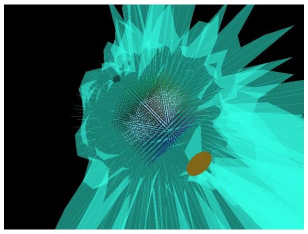

Figure 2: Charged particles located in CERN start to accelerate, generating EM radiation

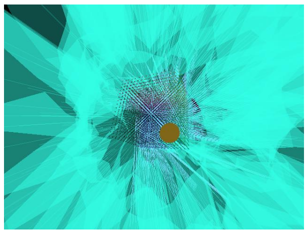

Figure 3: The Intensity increases further as the particles reach maximum velocity

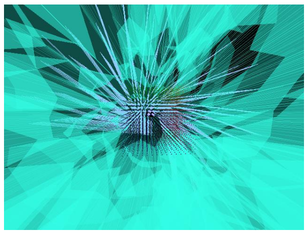

Figure 4: The integrity of the Spaceship can no longer tolerate the intense Electro-magnetic radiation and EXPLODES

## 4 Conclusion

The results achieved were not as expected. The motion of the accelerating charged particles can be modelled by a ring of current (Figure 2), thereby producing a Torus shaped Magnetic field. However, as can be seen in the above images, the field produced was not that of a Torus shape. Nonetheless, the method employed was indeed successful in detering the Alien invasion as can seen by the explosion in Figure 4.

## 5 Bibliography

”Introduction to Electrodynamics”, 3rd Edition, David J. Griffiths, Prentice-Hall Inc., 1999.

## A Simulation.c

```
#include <stdlib.h> /* malloc free */
#include <stdio.h>  /* fprintf */
#include <math.h>   /* sqrt */
#include <float.h>  /* FLTMAX */
#include "Simulation.h"

enum Animation { VOID, CURRENT, EXPLOSION };

/* constants */
const static float sol    = 3e8;
const static float mu0    = M_PI * 4e-7;
const static float scale  = .01;
const static float iscale = 1;
#define gradpi (16)
#define graddiv (pi / gradpi)
#define aliendiv (1536)         /* very simple alien made out of points */
const static float cutoff = .1; /* less then this won’t be shown */
const static float woverc = 1e9 / 3e8;

/* public class */
struct Simulation {
        void             (*vertex)(float, float, float);
        int              (*anim)(struct Simulation *, const int);
        enum Animation   num;
        float            rh_0, rh_1; /* the radii (rho) */
        float            d_ph;
        int              size;
        int              rsize;
        int              changed;
        float            alien[aliendiv];
        struct Component **grid;
        struct Polar     **gain;
};

/* private class */

struct Component {
        float p[3];   /* x, y, z */
        float I[3];   /* current */
        float colour; /* unused */
};

struct Polar {
        float p[2];    /* theta, phi */
        float norm[3]; /* cartesian */
        float A[3];    /* vector pot */
        float gain;
};

struct Component *Component(const float x, const float y, const float z);
void Component_(struct Component **cPtr);
struct Polar *Polar(const float theta, const float phi);
void Polar_(struct Polar **cPtr);
static void zero(float a[3]);
static void addto(float a[3], const float b[3]);
static void add(float a[3], const float b[3], const float c[3]);
static void cross(float a[3], const float b[3], const float c[3]);
static void distance(float d[3], const float a[3], const float b[3]);
static void mult(float p[3], const float a, const float b[3]);
static float square(const float a[3]);
static float mag2(const float a[3]);
void alien(float *a);
int blowup(struct Simulation *s, int frame);
int sim(struct Simulation *s, int frame);

/* public */

struct Simulation *Simulation(const int size, const int rsize, void (*v)(float, float, float)) {
        int i, x, y, z, theta, phi;
        float t;
        struct Simulation *s;

        /* fixme: size > maxint^(1/3) */
        if(size < 3 || size > 1625 || rsize < 3 || rsize > 1625 || !v) {
                fprintf(stderr, "Simulation: WTF?\n");
                return 0;
        }
        if(!(s = malloc(sizeof(struct Simulation) + sizeof(struct Component *) * size * size * size + sizeof(struct Polar *) * 2 * rsize * rsize))) {
                perror("Simulation constructor");
                Simulation_(&s);
                return 0;
        }
        s->vertex    = v;
        s->anim      = 0;
        s->num       = VOID;
        s->rh_0      = size;
        s->rh_1      = size * 1.1;
        s->d_ph      = (float)M_PI / rsize;
        s->size      = size;
        s->rsize     = rsize;
```

```
	s->changed   = -1;
	s->grid      = (struct Component **)(s + 1);
	s->gain      = (struct Polar **)(s->grid + size * size * size + 1);
	/* do the alien */
	alien(s->alien);
	/* assign null pointers */
	for(i = 0; i < size * size * s->size; i++) s->grid[i] = 0;
	for(i = 0; i < 2 * size * size;       i++) s->gain[i] = 0;
	fprintf(stderr, "Simulation: new %d, #%p.\n", size, (void *)s);
	/* alloc components */
	t = -(float)size / 2 + .5;
	for(z = 0; z < size; z++) {
		for(y = 0; y < size; y++) {
			for(x = 0; x < size; x++) {
				if(!(s->grid[z*size*size + y*size + x] = Component(t + x, t + y, t + z))) {
					Simulation_(&s);
					return 0;
				}
			}
		}
	}
	t = (float)M_PI / rsize;
	for(phi = 0; phi < rsize; phi++) {
		for(theta = 0; theta < 2*rsize; theta++) {
			if(!(s->gain[phi*2*rsize + theta] = Polar(t * 2 * theta, t * phi))) {
				Simulation_(&s);
				return 0;
			}
		}
	}

	return s;
}

void Simulation_(struct Simulation **simulationPtr) {
	int i;
	struct Simulation *simulation;

	if(!simulationPtr || !(simulation = *simulationPtr)) return;
	for(i = 0; i < simulation->size * simulation->size * simulation->size; i++) {
		if(simulation->grid[i]) Component_(&simulation->grid[i]);
	}
	for(i = 0; i < 2 * simulation->rsize * simulation->rsize; i++) {
		if(simulation->gain[i]) Polar_(&simulation->gain[i]);
	}
	fprintf(stderr, "~Simulation: erase, #%p.\n", (void *)simulation);
	free(simulation);
	*simulationPtr = simulation = 0;
}

int (*SimulationGetAnim(const struct Simulation *s))(struct Simulation *, const int) {
	if(!s) return 0;
	return s->anim;
}

void SimulationClearAnim(struct Simulation *s) {
	if(!s) return;
	s->anim = 0;
}

int SimulationGetSize(const struct Simulation *simulation) {
	if(!simulation) return 0;
	return simulation->size;
}

float SimulationGetColour(const struct Simulation *s, const int x, const int y, const int z) {
	if(!s) return 0;
	/*if(x >= size || y >= size || z >= size || x < 0 || y < 0 || z < 0) return 0;*/
	return s->grid[z*s->size*s->size + y*s->size + x]->colour;
}

void SimulationAnimation(struct Simulation *s) {
	if(!s) return;
	switch(s->num) {
		case VOID:
			printf("Aliens attack Earth! who will we turn to? CERN will save us!\n");
			printf("Passing two, oppositely-aligned electron beams with slight variations\n");
			printf("in the flux density, they create a AC current that powers a giant EMP\n");
			printf("tuned to the exact frequency of the alien battlecruisers resonance. Powering up.\n\n");
			s->anim = &sim;
			s->num = CURRENT;
			break;
		case CURRENT:
			printf("It worked!\n\n");
			s->anim = &blowup;
			s->num = EXPLOSION;
			break;
		case EXPLOSION:
			s->anim = 0;
			break;
		default:
			break;
	}
}

int SimulationUpdate(struct Simulation *s) {
	int size;          /* of the sim */
	int theta, phi, i; /* indices */
	float sphere[3], r[3], d, d2, si, co, ReTemp, ImTemp;
	float nr[3], Icrossnr[3], Re[3], Im[3], ReI[3], ImI[3], B[3];
	struct Component *c;
	struct Polar *p;
```

```
        if(!s) return 0;
        if(!s->changed) return -1;
        size = s->size;

        for(phi = 0; phi < s->rsize; phi++) {
                for(theta = 0; theta < 2*s->rsize; theta++) {
                        p = s->gain[phi*2*s->rsize + theta];
                        for(i = 0, zero(ReI), zero(ImI); i < size*size*size; i++) {
                                c = s->grid[i];
                                if(c->I[0] < cutoff && c->I[0] > -cutoff &&
                                   c->I[1] < cutoff && c->I[1] > -cutoff &&
                                   c->I[2] < cutoff && c->I[2] > -cutoff) continue;
                                /* r */
                                mult(sphere, s->rh_0, p->p); /* r' = rh_0 * n */
                                distance(r, c->p, sphere);   /* r  = r - r' */
                                d = sqrt(d2 = mag2(r));      /* d  = |r| */
                                mult(nr, 1/d, r);            /* nr = r/|r| */
                                /* J(i) x nr */
                                cross(Icrossnr, c->I, nr);
                                /* e^{-iwr/c}(1/d^2 - iw/c/d) */
                                si = sin(woverc*d);
                                co = cos(woverc*d);
                                ReTemp = co/d2 - woverc/d*si;
                                ImTemp = woverc/d*co - si/d2;
                                /* multipy them together */
                                mult(Re, ReTemp, Icrossnr);
                                mult(Im, ImTemp, Icrossnr);
                                /* 'integrate' */
                                addto(ReI, Re);
                                addto(ImI, Im);
                        }
                        /* approximate by rho_0 */
                        si = sin(woverc*s->rh_0);
                        co = cos(woverc*s->rh_0);
                        /* only need the real part: co ReI + i si i ImI */
                        mult(Re, co,  ReI);
                        mult(Im, -si, ImI);
                        add(B, Re, Im);
                        /* gain ~= |B|^2 */
                        p->gain = square(B);
                }
        }
        s->changed = 0;

        return -1;
}

int SimulationCurrent(const struct Simulation *s) {
        int i, size;
        struct Component *c;

        if(!s) return 0;
        size = s->size;
        for(i = 0; i < size*size*size; i++) {
                c = s->grid[i];
                if(c->I[0] < cutoff && c->I[1] < cutoff && c->I[2] < cutoff) continue;
                s->vertex(c->p[0], c->p[1], c->p[2]);
                s->vertex(c->p[0]+iscale*c->I[0], c->p[1]+iscale*c->I[1], c->p[2]+iscale*c->I[2]);
        }

        return -1;
}

int SimulationDraw(const struct Simulation *s) {
        int i;
        float rho/*, x, y, z*/;
        struct Polar *p;

        if(!s) return 0;
        rho = s->rh_0;
        for(i = 0; i < 2*s->rsize*s->rsize; i++) {
                p = s->gain[i];
                s->vertex(rho          *p->norm[0], rho          *p->norm[1], rho          *p->norm[2]);
                s->vertex((p->gain+rho)*p->norm[0], (p->gain+rho)*p->norm[1], (p->gain+rho)*p->norm[2]);
        }

        return -1;
}

int SimulationGain(const struct Simulation *s) {
        int theta, phi, i;
        float rho;
        struct Polar *p;
        if(!s) return 0;
        rho = s->rh_0;
        for(phi = 0; phi < s->rsize - 1; phi++) {
                for(i = 0; i <= s->rsize; i++) {
                        /* complete the ring by going back to the start */
                        theta = (i == s->rsize ? 0 : i);
                        p = s->gain[phi*2*s->rsize + theta];
                        s->vertex((p->gain+rho)*p->norm[0], (p->gain+rho)*p->norm[1], (p->gain+rho)*p->norm[2]);
                        p = s->gain[(phi+1)*2*s->rsize + theta];
                        s->vertex((p->gain+rho)*p->norm[0], (p->gain+rho)*p->norm[1], (p->gain+rho)*p->norm[2]);
                }
                /* add a degenerate vertex to preserve the winding order */
                p = s->gain[(phi+1)*2*s->rsize];
                s->vertex((p->gain+rho)*p->norm[0], (p->gain+rho)*p->norm[1], (p->gain+rho)*p->norm[2]);
        }
        return -1;
}
```

```
int SimulationAlien(const struct Simulation *s) {
        int i;
        for(i = 0; i < sizeof(s->alien) / sizeof(float); i += 3) {
                s->vertex(s->alien[i], s->alien[i+1], s->alien[i+2]);
        }
        return -1;
}

/* private class */

struct Component *Component(const float x, const float y, const float z) {
        struct Component *c;

        if(!(c = malloc(sizeof(struct Component)))) {
                perror("Component constructor");
                Component_(&c);
                return 0;
        }
        c->p[0] = x; c->p[1] = y; c->p[2] = z;
        c->I[0] = c->I[1] = c->I[2] = 0;
        c->colour = (float)rand() / RAND_MAX; /* not used */
        /*spam fprintf(stderr, "Component: new (%f,%f,%f) #%p.\n", c->p[0], c->p[1], c->p[2], (void *)c);*/

        return c;
}

void Component_(struct Component **cPtr) {
        struct Component *c;

        if(!cPtr || !(c = *cPtr)) return;
        /*spam fprintf(stderr, "~Component: erase, #%p.\n", (void *)c);*/
        free(c);
        *cPtr = c = 0;
}

struct Polar *Polar(const float theta, const float phi) {
        struct Polar *p;

        if(!(p = malloc(sizeof(struct Polar)))) {
                perror("Polar constructor");
                Polar_(&p);
                return 0;
        }
        p->p[0] = theta; p->p[1] = phi;
        p->norm[0] = sin(phi) * cos(theta);
        p->norm[1] = sin(phi) * sin(theta);
        p->norm[2] = cos(phi);
#if 0
        p->xunit[0] =  cos(phi) * sin(theta); /* rho */
        p->xunit[1] =  cos(phi) * cos(theta); /* theta */
        p->xunit[2] = -sin(phi);              /* phi */
        p->oversin_th = (sin(theta) == 0 ? FLT_MAX : 1 / sin(theta));
        p->sin_th = sin(theta); p->cos_th = cos(phi);
#endif
        p->A[0] = p->A[1] = p->A[2] = 0;
        p->gain = 0;
        /*spam fprintf(stderr, "Polar: new #%p.\n", (void *)p);*/

        return p;
}

void Polar_(struct Polar **pPtr) {
        struct Polar *p;

        if(!pPtr || !(p = *pPtr)) return;
        /*spam fprintf(stderr, "~Polar: erase, #%p.\n", (void *)p);*/
        free(p);
        *pPtr = p = 0;
}

/* private */

static void zero(float a[3]) {
        a[0] = a[1] = a[2] = 0;
}

static void addto(float a[3], const float b[3]) {
        a[0] += b[0];
        a[1] += b[1];
        a[2] += b[2];
}

static void add(float a[3], const float b[3], const float c[3]) {
        a[0] = b[0] + c[0];
        a[1] = b[1] + c[1];
        a[2] = b[2] + c[2];
}

static void cross(float a[3], const float b[3], const float c[3]) {
        a[0] = b[1]*c[2] + b[2]*c[1];
        a[1] = b[2]*c[0] + b[0]*c[2];
        a[2] = b[0]*c[1] + b[1]*c[0];
}

static void distance(float d[3], const float a[3], const float b[3]) {
        d[0] = b[0] - a[0];
        d[1] = b[1] - a[1];
        d[2] = b[2] - a[2];
}

static void mult(float p[3], const float a, const float b[3]) {
        p[0] = a * b[0];
        p[1] = a * b[1];
        p[2] = a * b[2];
}

static float square(const float a[3]) {
        return a[0]*a[0] + a[1]*a[1] + a[2]*a[2];
}

static float mag2(const float a[3]) {
        return a[0]*a[0] + a[1]*a[1] + a[2]*a[2];
}

/** old-school */
void alien(float *a) {
        int n = 0;
        float theta, phi;
        for(phi = M_PI / 32; phi < M_PI; phi += M_PI / 16) {
                for(theta = M_PI / 32; theta < 2*M_PI; theta += M_PI / 16) {
                        a[n++] = 3. * sin(phi) * cos(theta);
                        a[n++] = 3. * sin(phi) * sin(theta);
                        a[n++] = 1. * cos(phi);
                }
        }
        fprintf(stderr, "Number of points_alien %d.\n", n);
}

int blowup(struct Simulation *s, int frame) {
        int i;
        for(i = 0; i < sizeof(s->alien) / sizeof(float); i++) {
                s->alien[i] += 0.05 * s->alien[i] + .1 * (.5 - (float)rand() / RAND_MAX);
        }
        if(frame > 256) return 0;
        return -1;
}

int sim(struct Simulation *s, int frame) {
        int x, y, z, size = s->size;
        struct Component *c;
        /* define a current */
        /*for(z = 0; z < size; z++) s->grid[z*size*size + size*size>>1 + size>>1]->I[2] = 10.5;*/
        for(z = 0; z < size; z++) {
                for(y = 0; y < size; y++) {
                        for(x = 0; x < size; x++) {
                                c = s->grid[z*size*size + y*size + x];
                                /*temp = 3 - sqrt(c->p[0]*c->p[0] + c->p[1]*c->p[1]);
                                if(temp*temp + c->p[3]*c->p[3] > 2) continue; torus, no? */
                                if(c->p[0]*c->p[0] + c->p[1]*c->p[1] + c->p[2]*c->p[2] > 25) continue;
                                c->I[0] =  (float)c->p[1] * (float)frame*.1;
                                c->I[1] = -(float)c->p[0] * (float)frame*.1;
                                c->I[2] = 0;
                        }
                }
        }
        s->changed = -1;
        if(frame > 64) SimulationAnimation(s);
        return -1;
}
```

## B Simulation.h

```
struct Simulation;

struct Simulation *Simulation(const int size, const int rsize, void (*v)(float, float, float));
void Simulation_(struct Simulation **simulationPtr);
int (*SimulationGetAnim(const struct Simulation *s))(struct Simulation *, const int);
void SimulationClearAnim(struct Simulation *s);
int SimulationGetSize(const struct Simulation *simulation);
float SimulationGetColour(const struct Simulation *s, const int x, const int y, const int z);
void SimulationAnimation(struct Simulation *s);
int SimulationUpdate(struct Simulation *s);
int SimulationCurrent(const struct Simulation *s);
int SimulationDraw(const struct Simulation *s);
int SimulationGain(const struct Simulation *s);
int SimulationAlien(const struct Simulation *s);
```

## C Open.c

```
#include <stdlib.h> /* malloc free */
#include <stdio.h>  /* fprintf */
#ifdef GL
#include <GL/gl.h>
#include <GL/glut.h>
#else
#include <OpenGL/gl.h> /* OpenGL ** may be GL/gl.h */
#include <GLUT/glut.h> /* GLUT */
#endif
#include "Simulation.h"

struct Open *Open(const int width, const int height, const char *title);
void Open_(void);
void update(int);
void display(void);
void resize(int width, int height);
void keyUp(unsigned char k, int x, int y);
void keyDn(unsigned char k, int x, int y);
void keySpecial(int key, int x, int y);
void cube(const float x, const float y, const float z, const float a);

struct Open {
        struct Simulation *s;
        float             rot;
        int               frame;
};

struct Open *open = 0;

const static int granularity   = 16;
const static int angulargranularity = 32;
const static float speed       = .7;
static float black_of_space[4] = { 0, 0, 0, 0 }/*{ 1, 1, 1, 0 }*/;
static float current[4]        = { .7, .7, 1, 1 };
static float cool[4]           = { .2, 1, .9, .3 };

int main(int argc, char **argv) {
        /* negotiate with library */
        glutInit(&argc, argv);

        if(!Open(640, 480, "Simuation")) return EXIT_FAILURE;
        /* atexit because the loop never returns */
        if(atexit(&Open_)) perror("~Open");
        glutMainLoop();

        return EXIT_SUCCESS;
}

struct Open *Open(const int width, const int height, const char *title) {
        GLfloat lightPos[4] = { 1.0, 10.0, 10.0, .5 }, lightAmb[4] = { 1, .5, .2, 1 };

        if(open || width <= 0 || height <= 0 || !title) {
                fprintf(stderr, "Open: error initialising.\n");
                return 0;
        }
        if(!(open = malloc(sizeof(struct Open)))) {
                perror("Open constructor");
                Open_();
                return 0;
        }
        open->s     = 0;
        open->rot   = 0;
        open->frame = 0;
        if(!(open->s = Simulation(granularity, angulargranularity, &glVertex3f))) {
                Open_();
                return 0;
        }
        fprintf(stderr, "Open: new, #%p.\n", (void *)open);
        /* initial conditions */
        glutInitDisplayMode(GLUT_DOUBLE/* | GLUT_DEPTH*/); /* RGB[A] is implied */
        glutInitWindowSize(width, height); /* just a suggestion */
        /* create */
        glutCreateWindow(title);
        /* initialise */
        glShadeModel(GL_SMOOTH);
        glClearColor(black_of_space[0], black_of_space[1], black_of_space[2], black_of_space[3]);
        /* z-buffer! */
        /*glEnable(GL_DEPTH_TEST);
        glClearDepth(1.0);
        glDepthFunc(GL_LEQUAL);*/
        glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
        glEnable(GL_NORMALIZE);
        glEnable(GL_COLOR_MATERIAL);
        glEnable(GL_POINT_SMOOTH);
        /* lighing */
        /*glEnable(GL_LIGHTING);*/
        glEnable(GL_LIGHT0);
        glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
        glLightModelfv(GL_LIGHT_MODEL_AMBIENT, (GLfloat *)&lightAmb);
        /* blending */
        glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        /*glShadeModel(GL_SMOOTH);*/
        /* set callbacks */
        glutDisplayFunc(&display);
        glutReshapeFunc(&resize);
        glutKeyboardFunc(&keyDn);
        /* glutIdleFunc(0); disable */
        glutTimerFunc(25, update, 0);
        /* just to be shure */
        glutReshapeWindow(width, height);

        return open;
}

void Open_(void) {
        if(!open) return;
        fprintf(stderr, "~Open: erase, #%p.\n", (void *)open);
        if(open->s) Simulation_(&open->s);
        free(open);
        open = 0;
}

/* private */

void update(int value) {
        int (*e)(struct Simulation *, int);
        open->rot += speed;
        glutPostRedisplay();
        glutTimerFunc(25, update, 0);
        SimulationUpdate(open->s);
        if((e = SimulationGetAnim(open->s))) {
                if((e(open->s, open->frame))) {
                        open->frame++;
                } else {
                        SimulationClearAnim(open->s);
                        open->frame = 0;
                }
        }
}

void display(void) {
        float size;
        GLfloat x, y, z, c;

        size   = SimulationGetSize(open->s);

        /* clear screen and depthbuf, make sure it's modelview, and reset the matrix */
        glClear(GL_COLOR_BUFFER_BIT/* | GL_DEPTH_BUFFER_BIT*/);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glTranslatef(.0*size, .0*size, -1.9 * size);
        glRotatef(open->rot, open->rot+60, open->rot+30, 1);
        glBegin(GL_LINES);
        glColor4f(current[0], current[1], current[2], current[3]);
        SimulationCurrent(open->s);
        glColor4f(cool[0],    cool[1],    cool[2],    cool[3]);
        SimulationDraw(open->s);
        glEnd();
        glBegin(GL_TRIANGLE_STRIP);
        SimulationGain(open->s);
        glEnd();
        for(x = -size/2+.5; x < size/2; x += 1) {
                for(y = -size/2+.5; y < size/2; y += 1) {
                        for(z = -size/2+.5; z < size/2; z += 1) {
                                c = .5;/*SimulationGetColour(open->s, x, y, z);*/
                                glPointSize(c + 1);
                                glColor4f((float)x/size, (float)y/size, (float)z/size, 1.);
                                glBegin(GL_POINTS);
                                glVertex3f(x + .5, y + .5, z + .5);
                                glEnd();
                        }
                }
        }
        glTranslatef(0, 0, size);
        glPointSize(10.5);
        glColor4f(.5, .4, .1, 1.);
        glBegin(GL_POINTS);
        SimulationAlien(open->s);
        glEnd();
        glutSwapBuffers();
}

void resize(int width, int height) {
        if(width <= 0 || height <= 0) return;
        glViewport(0, 0, width, height);
        /* calculate the projection */
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(90., (float)width / height, 5., 200.);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glTranslatef(0.0, 0.0, -500/*-23.0 / 2.5*/);
}

void keyDn(unsigned char k, int x, int y) {
        SimulationAnimation(open->s);
}
```