Magnetic Simulation of Exploding Planet by Death Star

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

2009-02-15

Abstract

The Gauss-Seidel iterative algorithm was used to simulate the magnetic field on a constant grid. Each grid point has a possibly different permeability, this allows magnetism to propagate through different mediums. We used it to simulate a planet much like the Earth, represented by a rotating molten magma core and a surrounding region of Olivine. This planet is then hit by a positron beam from the Death Star which destroys it.

1 Introduction

The Death Star is the ultimate weapon of terror in the universe. Modeling the Death Star as a sphere of iron, we set out to not only map the interacting magnetic fields of Earth, but also that of the Death Star as it orbits at a distance. Unlike the Earth, the Death Star, given a value of μ_{Fe}, does not generate a magnetic field, so much as it is permeated by that of the Earth.

2 Discussion

The simulation starts off with no current, and an isotropic μ_0 spread out over the volume. The magma is then generated with a circular current simulating the core of Earth. Olivine is added to form the mantle and this changes the magnetic field because $\mu \neq \mu_0$. The Death Star appears, having μ_{Fe}. Our simulation also involves a positron beam which destroys all of human life. Evidently, this beam will generate a magnetic field, as it constitutes a large current.

3 Theory

In general, the equation linking the magnetic vector potential, $A(r)$, and the associated current is given by Equation 1[1].

$$\nabla \times (\frac{1}{\mu} \nabla \times A) = J \tag{1}$$

This equation was used to find the vector potential from the model currents. Next, we need to model the magnetic field proper. This was done by using Equation 2[1].

$$B = \nabla \times A \tag{2}$$

In order to implement this, we need a grid system to attach a vectorial value at each point of space in our given grid. We used a rudimentary simple grid (without quadtrees.) Then we used the Gauss-Siedel[4] algorithm in order to evaluate the values at each point in our three-space. In general, the Gauss-Siedel algorithm can be expressed as in Equation 3.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right), \; i = 1, 2, \ldots, n. \tag{3}$$

A relaxation factor of 1.9 was used for the computation. It was found that this gave good convergence.

The simulated Earth-like planet was a core of iron with a μ of 5000[5]. We found our model converged very slowly with such an extreme magnetic permeability, so we massaged it accordingly. The mantle was a layer of Olivine[2][3]; again, our model diverged with so low a value, so we set it to 0.3.

The current around the boundary was zero and μ_r was one. Magnetic fields were not calculated at the boundary; this gave $A(\text{boundary}) = \vec{0}$ as a boundary condition, consistent with space.

A Monte Carlo algorithm was used to update the points. We didn't notice a difference at 32 grid granularity.

4 Results

The μ_r values that were used in the programming were 0.3 for Olivine, 0.9 for iron and magma, and 4 for plasma. These were determined to produce a convergence in the Gauss-Seidel algorithm. The graphics engine assigned a value of $1 - \mu$ to the colour and the size of the point.

	Error	Frame	Zero
Space	0.00	0	0
Planet Core	36.15	1	58
Planet Crust	0.69	0	50
Death Star	0.00	0	0
Poof	9486.97	27	117

Table 1: Normalised maximum average errors, the frame that it happens, and the frame that the errors to go to zero. The Death Star was too far away from the planet to have an effect on it's magnetic field.

As seen in Table 4, the maximum variances were made by introducing new elements into the simulation, as expected. These variances are exact because the simulation was the same every time. The 'Poof' phase, values were not static, the Earth was blowing up, so high values were expected.

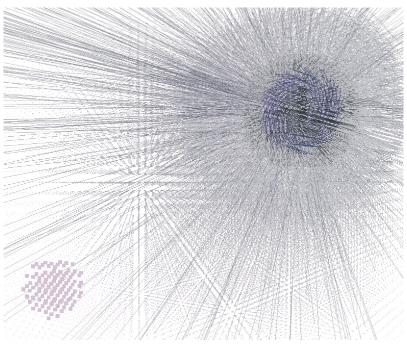

Figure 1: Emergence of Death Star.

We see in Figure 1 the station arriving, entering Earth's orbit. Notice the magnetic lines emanating from the Earth produced by the geodynamo.

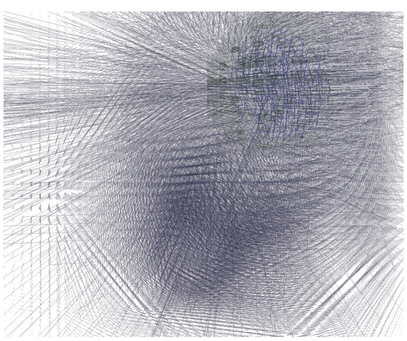

Figure 2: Positron beam from Death Star.

In Figure 2, after Grand Moff Tarkin gives the go-ahead, the positron beam is unleashed on the unsuspecting population of Earth.

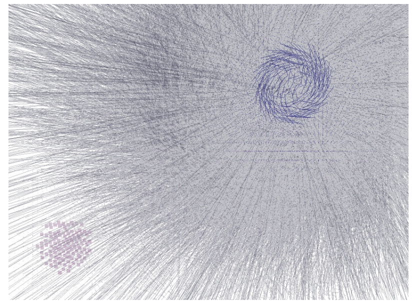

Figure 3: Planet exploding.

In Figure 3 we see the plasma remnants of the Earth. This was taken just before the field created by the vaporised magma dissipated thought space.

5 Conclusion

According to preliminary results, it shows that the Earth cannot survive a positron beam attack. The magnetic field were successfully generated from the modeled currents with adequate precision. With modified μ values to prevent divergence, the Gauss-Seidel iteration method proved reasonably successful.

References

[1] DJ Griffiths. Introduction to Electrodynamics, Third Edition. Prentice Hall, 1999.

[2] HP Gunnlaugssona, O Helgasonb, L Kristjánssonb, P Nørnbergc, H Rasmussenc, S Steinpórssonb, and G Weyera. Magnetic properties of olivine basalt: Application to mars. Physics of The Earth and Planetary Interiors, 154(3-4):276–289, March 2006.

[3] Y Iwa, T Nakamura, Y Yamada, M Tabuchi, and H Kageyama. Magnetic study on olivine compounds. Trans Mater Res Soc Jpn, 29(4):1683–1686, 2004.

[4] JD Jackson. Classical Electrodynamics, Second Edition. Wiley, 1975.

[5] Taylor, Parker, and Lengenberg. American Institute of Physics Handbook Third Edition. McGraw-Hill, 1972.

A Simulation.c

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

/* \mu_r values; incorrect, but these value lead to convergence */
static const float olivine = 0.3, iron = .9, magma = .9, plasma = 4, space = 1;
static const float rel = 1.9; /* rate at which new values are preferred */
static const float ib  = .03; /* relation current-magnetic */

enum Animation { SPACE, INNER, PLANET, DEATH_STAR, BEAM_DOOM, POOF };

/* public class */
struct Simulation {
	int              size;
	struct Component **grid;
	void             (*vertex)(float, float, float);
	struct Component **montecarlo;
	enum Animation   animation;
	int              (*explode)(struct Simulation *, const int);
	int              t;
	int              err2maxt;
	float            err2max;
	float planet[3], death[3];
};

/* private class */
struct Component {
	float x[3];       /* position */
	float a[3];       /* vector potential */
	float b[3];       /* magnetic field */
	float current[3]; /* current density */
	float mu;         /* mu_r -- property of the material */
};

struct Component *Component(const float x, const float y, const float z);
void Component_(struct Component **cPtr);
void sphere(struct Simulation *s, const float p[3], const int r2inner, const int r2, const float cross, const float mu);
void beamdoom(const struct Simulation *s, const int coof);
int blowup(struct Simulation *s, int frame);
int montecarlo(struct Simulation *s);

/* public */

struct Simulation *Simulation(int size, void (*v)(float, float, float)) {
	int i, x, y, z;
	struct Simulation *simulation;

	/* fixme: size > maxint^(1/3) */
	if(size < 3 || size > 1625 || !v) {
		fprintf(stderr, "Simulation: need more info.\n");
		return 0;
	}
	if(!(simulation = malloc(sizeof(struct Simulation) + sizeof(struct Component *) * size * size * size))) {
		perror("Simulation constructor");
		Simulation_(&simulation);
		return 0;
	}
	simulation->size      = size;
	simulation->grid      = (struct Component **)(simulation + 1);
	for(i = 0; i < size * size * simulation->size; i++) simulation->grid[i] = 0;
	simulation->vertex     = v;
	simulation->montecarlo = 0;
	simulation->animation  = SPACE;
	simulation->explode    = 0;
	simulation->t          = 0;
	simulation->err2maxt   = 0;
	simulation->err2max    = 0;
	simulation->planet[0]  = .30 * size;
	simulation->planet[1]  = .70 * size;
	simulation->planet[2]  = .70 * size;
	simulation->death[0]   = .85 * size;
	simulation->death[1]   = .15 * size;
	simulation->death[2]   = .15 * size;
	fprintf(stderr, "Simulation: new %d, #%p.\n", size, (void *)simulation);
	/* alloc components */
	for(z = 0; z < size; z++) {
		for(y = 0; y < size; y++) {
			for(x = 0; x < size; x++) {
				if(!(simulation->grid[z*size*size + y*size + x] = Component(x, y, z))) {
					Simulation_(&simulation);
					return 0;
				}
			}
		}
	}
	if(!montecarlo(simulation)) {
		Simulation_(&simulation);
		return 0;
	}

	return simulation;
}

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

	if(!simulationPtr || !(simulation = *simulationPtr)) return;
	if(simulation->montecarlo) {
```

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

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

float SimulationGetMu(const struct Simulation *s, const int x, const int y, const int z) {
	if(!s || x < 0 || y < 0 || z < 0 || x >= s->size || y >= s->size || z >= s->size) return 1.;
	return s->grid[z*s->size*s->size + y*s->size + x]->mu;
}

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

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

int SimulationUpdate(struct Simulation *s) {
	int   x,  y,  z,  n;                      /* loop indecies */
	float ax, ay, az;                         /* var temp, c->a */
	float err2 = 0;                           /* error^2 (ax - c->a[0]) */
	float oneover6mu, dmu, mu;                /* mu temp */
	float mux1, mux2, muy1, muy2, muz1, muz2; /* mu vars */
	struct Component *c, *xm, *xp, *ym, *yp, *zm, *zp; /* (p)lus, (m)inus */

	if(!s) return 0;

	/* \nabla x (1/u \nabla x A) = J */

	/* monte carlo array */
	for(n = 0; (c = s->montecarlo[n]); n++) {
		x = c->x[0];
		y = c->x[1];
		z = c->x[2];
		/* components this, right, . . . fixme: anti-alised samples */
		xp = s->grid[z       * s->size * s->size + y * s->size       + (x + 1)];
		xm = s->grid[z       * s->size * s->size + y * s->size       + (x - 1)];
		yp = s->grid[z       * s->size * s->size + (y + 1) * s->size + x];
		ym = s->grid[z       * s->size * s->size + (y - 1) * s->size + x];
		zp = s->grid[(z + 1) * s->size * s->size + y       * s->size + x];
		zm = s->grid[(z - 1) * s->size * s->size + y       * s->size + x];
		/* mu, fixme: optimised by pre-computing */
		oneover6mu = 1 / (6 * c->mu);
		dmu        = (xp->mu - xm->mu) * oneover6mu;
		mux1 = 1 - dmu;
		mux2 = 1 + dmu;
		dmu        = (yp->mu - ym->mu) * oneover6mu;
		muy1 = 1 - dmu;
		muy2 = 1 + dmu;
		dmu        = (zp->mu - zm->mu) * oneover6mu;
		muz1 = 1 - dmu;
		muz2 = 1 + dmu;
		/* calculate a */
		mu = mux1*xp->a[0] + mux2*xm->a[0] + muy1*yp->a[0] + muy2*ym->a[0] + muz1*zp->a[0] + muz2*zm->a[0];
		ax = (rel / 6) * (mu + c->current[0] * c->mu) + (1 - rel) * c->a[0];
		mu = mux1*xp->a[1] + mux2*xm->a[1] + muy1*yp->a[1] + muy2*ym->a[1] + muz1*zp->a[1] + muz2*zm->a[1];
		ay = (rel / 6) * (mu + c->current[1] * c->mu) + (1 - rel) * c->a[1];
		mu = mux1*xp->a[2] + mux2*xm->a[2] + muy1*yp->a[2] + muy2*ym->a[2] + muz1*zp->a[2] + muz2*zm->a[2];
		az = (rel / 6) * (mu + c->current[2] * c->mu) + (1 - rel) * c->a[2];
		/* error^2 */
		err2 += (ax - c->a[0]) * (ax - c->a[0]) + (ay - c->a[1]) * (ay - c->a[1]) + (az + c->a[2]) * (az + c->a[2]);
		/* assign */
		c->a[0] = ax;
		c->a[1] = ay;
		c->a[2] = az;
		/* calculate b (z/dy - y/dz, x/dz - z/dx, y/dx - x/dy) */
		c->b[0] = -((zp->a[1] - zm->a[1]) - (yp->a[2] - ym->a[2])) / 2;
		c->b[1] = -((xp->a[2] - xm->a[2]) - (zp->a[0] - zm->a[0])) / 2;
		c->b[2] = -((yp->a[0] - ym->a[0]) - (xp->a[1] - xm->a[1])) / 2;
	}
	if(err2 > s->err2max) { s->err2max = err2; s->err2maxt = s->t; }
	if(err2 > 1) s->t++;
	return -1;
}

int SimulationCurrent(const struct Simulation *s) {
	int x, y, z;
	struct Component *v;

	if(!s) return 0;

	for(z = 0; z < s->size; z++) {
		for(y = 0; y < s->size; y++) {
			for(x = 0; x < s->size; x++) {
				v = s->grid[z * s->size * s->size + y * s->size + x];
				s->vertex(x + .5,                    y + .5,                    z + .5);
				s->vertex(x + .5 + ib*v->current[0], y + .5 + ib*v->current[1], z + .5 + ib*v->current[2]);
			}
```

```
                }
        }

        return -1;
}

int SimulationMagnetic(const struct Simulation *s) {
        int x, y, z;
        struct Component *v;

        if(!s) return 0;

        for(z = 0; z < s->size; z++) {
                for(y = 0; y < s->size; y++) {
                        for(x = 0; x < s->size; x++) {
                                v = s->grid[z * s->size * s->size + y * s->size + x];
                                if(-.1 < v->b[0] && v->b[0] < .1 &&
                                   -.1 < v->b[1] && v->b[1] < .1 &&
                                   -.1 < v->b[2] && v->b[2] < .1) continue;
                                s->vertex(x + .5,           y + .5,           z + .5);
                                s->vertex(x + .5 + v->b[0], y + .5 + v->b[1], z + .5 + v->b[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->x[0] = x;
        c->x[1] = y;
        c->x[2] = z;
        c->a[0]       = c->a[1]       = c->a[2]       = 0;
        c->b[0]       = c->b[1]       = c->b[2]       = 0;
        c->current[0] = c->current[1] = c->current[2] = 0;     /* current density */
        c->mu                                         = space; /* mu_r */
        /*spam fprintf(stderr, "Component: new #%p.\n", (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;
}

/** animation fn */
void SimulationAnimation(struct Simulation *s) {
        int a = s->size;

        printf("maximum average error %f at frame %d; took %d to die out; ", s->err2max / (a*a*a), s->err2maxt, s->t);
        s->t        = 0;
        s->err2max  = 0;
        s->err2maxt = 0;

        switch(s->animation) {
                case SPACE:
                        printf("inner\n");
                        s->animation = INNER;
                        sphere(s, s->planet, 0, 0.5 * a, .8 * a, magma);
                        break;
                case INNER:
                        printf("planet\n");
                        s->animation = PLANET;
                        sphere(s, s->planet, 0.5 * a, 0.8 * a, 0, olivine);
                        break;
                case PLANET:
                        printf("death star\n");
                        s->animation = DEATH_STAR;
                        sphere(s, s->death, 0, .3 * a, 0, iron);
                        break;
                case DEATH_STAR:
                        printf("beam death\n");
                        s->animation = BEAM_DOOM;
                        beamdoom(s, 1);
                        break;
                case BEAM_DOOM:
                        printf("poof\n");
                        s->animation = POOF;
                        s->explode   = &blowup;
                        beamdoom(s, -1);
                        break;
                case POOF:
                        printf("that's all\n");
                        break;
                default:
                        printf("WTH?");
                        s->animation = SPACE;
                        break;
```

```
	}
}

/* private */

/** animation */
void sphere(struct Simulation *s, const float p[3], const int r2inner, const int r2, const float cross, const float mu) {
	int   x,  y,  z, a = s->size;
	float rx, ry, rz, d2;
	struct Component *c;

	for(z = 0; z < a; z++) {
		for(y = 0; y < a; y++) {
			for(x = 0; x < a; x++) {
				c  = s->grid[z*a*a + y*a + x];
				rx = x - p[0];
				ry = y - p[1];
				rz = z - p[2];
				d2 = rx*rx + ry*ry + rz*rz;
				if(d2 < r2 && d2 >= r2inner) {
					c->mu = mu;
					c->current[0] =  ry * cross;
					c->current[1] = -rx * cross;
				}
			}
		}
	}
}

/** animation */
void beamdoom(const struct Simulation *s, const int coof) {
	const int step = 32;
	int i, a = s->size;
	float p[3];
	struct Component *c;

	/* start at the death star */
	p[0] = s->death[0];
	p[1] = s->death[1];
	p[2] = s->death[2];
	for(i = 0; i < step; i++) {
		p[0] += (s->planet[0] - s->death[0]) / step;
		p[1] += (s->planet[1] - s->death[1]) / step;
		p[2] += (s->planet[2] - s->death[2]) / step;
		c = s->grid[(int)p[2]*a*a + (int)p[1]*a + (int)p[0]];
		c->current[0] += coof * (s->planet[0] - s->death[0]) * 3;
		c->current[1] += coof * (s->planet[1] - s->death[1]) * 3;
		c->current[2] += coof * (s->planet[2] - s->death[2]) * 3;
	}
}

/** return 0 for finished */
int blowup(struct Simulation *s, const int frame) {
	int x, y, z, a = s->size;
	float rx, ry, rz, d, d2;
	struct Component *c;

	if(frame < 20) {
		for(z = 0; z < s->size; z++) {
			for(y = 0; y < s->size; y++) {
				for(x = 0; x < s->size; x++) {
					c  = s->grid[z*a*a + y*a + x];
					rx = x - s->planet[0];
					ry = y - s->planet[1];
					rz = z - s->planet[2];
					d2 = rx*rx + ry*ry + rz*rz;
					d  = sqrt(d2);
					if(d < .02 * frame * a) {
						/* plasma */
						c->current[0] += (float)rand() / RAND_MAX * .5;
						c->current[1] += (float)rand() / RAND_MAX * .5;
						c->current[2] += (float)rand() / RAND_MAX * .5;
						c->mu = plasma;
					}
				}
			}
		}
	} else if(frame < 40) {
		for(z = 0; z < s->size; z++) {
			for(y = 0; y < s->size; y++) {
				for(x = 0; x < s->size; x++) {
					c  = s->grid[z*a*a + y*a + x];
					rx = x - s->planet[0];
					ry = y - s->planet[1];
					rz = z - s->planet[2];
					d2 = rx*rx + ry*ry + rz*rz;
					d  = sqrt(d2);
					if(d < .02 * (frame - 20) * a) {
						c->current[0] = c->current[1] = c->current[2] = 0;
						c->mu = space;
					}
				}
			}
		}
	} else {
		return 0;
	}
	return -1;
}

/** initalise Monte Carlo array */
int montecarlo(struct Simulation *s) {
```

```
	int i, x, y, z, a = s->size, m = 0;
	int size = (a-2)*(a-2)*(a-2);
	struct Component **monte, *temp;

	if(s->montecarlo) return 0;
	if(!(monte = malloc(sizeof(struct Component *) * (size + 1)))) {
		perror("Monte Carlo");
		return 0;
	}
	/* start with them in order */
	for(z = 1; z < a - 1; z++) {
		for(y = 1; y < a - 1; y++) {
			for(x = 1; x < a - 1; x++) {
				monte[m++] = s->grid[z*a*a + y*a + x];
			}
		}
	}
	monte[m] = 0;
	/* mess with the order Monaco style */
	for(i = 0; i < size; i++) {
		if(i == (m = rand() % size)) continue;
		/*monte[m] ^= monte[i] ^= monte[m] ^= monte[i];*/
		temp = monte[m];
		monte[m] = monte[i];
		monte[i] = temp;
	}

	s->montecarlo = monte;

	return -1;
}
```

B Simulation.h

```
struct Simulation;

struct Simulation *Simulation(int size, void (*v)(float, float, float));
void Simulation_(struct Simulation **simulationPtr);
int SimulationGetSize(const struct Simulation *simulation);
float SimulationGetMu(const struct Simulation *s, const int x, const int y, const int z);
int (*SimulationGetExplode(const struct Simulation *s))(struct Simulation *, const int);
void SimulationClearExplode(struct Simulation *s);
int SimulationUpdate(struct Simulation *s);
int SimulationCurrent(const struct Simulation *s);
int SimulationMagnetic(const struct Simulation *s);
void SimulationAnimation(struct Simulation *s);
```

C Open.c

```
#include <stdlib.h> /* malloc free */
#include <stdio.h>  /* fprintf */
#include <OpenGL/gl.h> /* OpenGL ** may be GL/gl.h */
#include <GLUT/glut.h> /* GLUT */
#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   = 32;
const static float speed       = .7;
static float black_of_space[4] = { 0, 0, 0, 0 }/*{ 1, 1, 1, 0 }*/;
static float current[4]        = { 0, 0, 1, 1 }/*{ .4, .4, .7, 1 }*/;
static float magnetic[4]       = { .2, 1, .9, .3 }/*{ .2, .2, .3, .3 }*/;

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

	if(!Open(360, 240, "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, 0.0 }, 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, &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]);
	glClearDepth(1.0);
	/*glEnable*/glDisable(GL_DEPTH_TEST);
	glDepthFunc(GL_LEQUAL);
	glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
	glEnable(GL_DEPTH_TEST);
	/*glEnable(GL_LIGHTING);*/
	glEnable(GL_LIGHT0);
	glEnable(GL_NORMALIZE);
	glEnable(GL_COLOR_MATERIAL);
	glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
	glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
	glLightModelfv(GL_LIGHT_MODEL_AMBIENT, (GLfloat *)&lightAmb);
	glShadeModel(GL_FLAT);
	glutReshapeWindow(width, height);
	/* set callbacks */
	glutDisplayFunc(&display);
	glutReshapeFunc(&resize);
	glutKeyboardFunc(&keyDn);
	/* glutIdleFunc(0); disable */
	glutTimerFunc(25, update, 0);

	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 = SimulationGetExplode(open->s))) {
		if((e(open->s, open->frame))) {
			open->frame++;
		} else {
			SimulationClearExplode(open->s);
			open->frame = 0;
		}
	}
}

void display(void) {
	int size;
	GLfloat x, y, z, a, offset;

	/* 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();
	size   = SimulationGetSize(open->s);
	offset = -(float)size / 2 + .5;
	glTranslatef(0, 0, 3 * offset);
	glRotatef(open->rot, 0, 1, 0);
	glTranslatef(offset, offset + .2, offset);
	glBegin(GL_LINES);
	glColor4f(current[0], current[1], current[2], current[3]);
	SimulationCurrent(open->s);
	glColor4f(magnetic[0], magnetic[1], magnetic[2], magnetic[3]);
	SimulationMagnetic(open->s);
	glEnd();
	for(x = 0; x < size; x += 1) {
		for(y = 0; y < size; y += 1) {
			for(z = 0; z < size; z += 1) {
				a = 1 - SimulationGetMu(open->s, x, y, z);
				glPointSize(a * 64 + 1);
				/*glColor4f(.5*(1 - a), .2, .5*(1 - a), .3);*/
				glColor4f(1. - a, 5., 1. - a, .3);
```

```
                                glBegin(GL_POINTS);
                                glVertex3f(x + .5, y + .5, z + .5);
                                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(45.0f,(float)width / height, 0.1f, 100.0f);
        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);
}
```