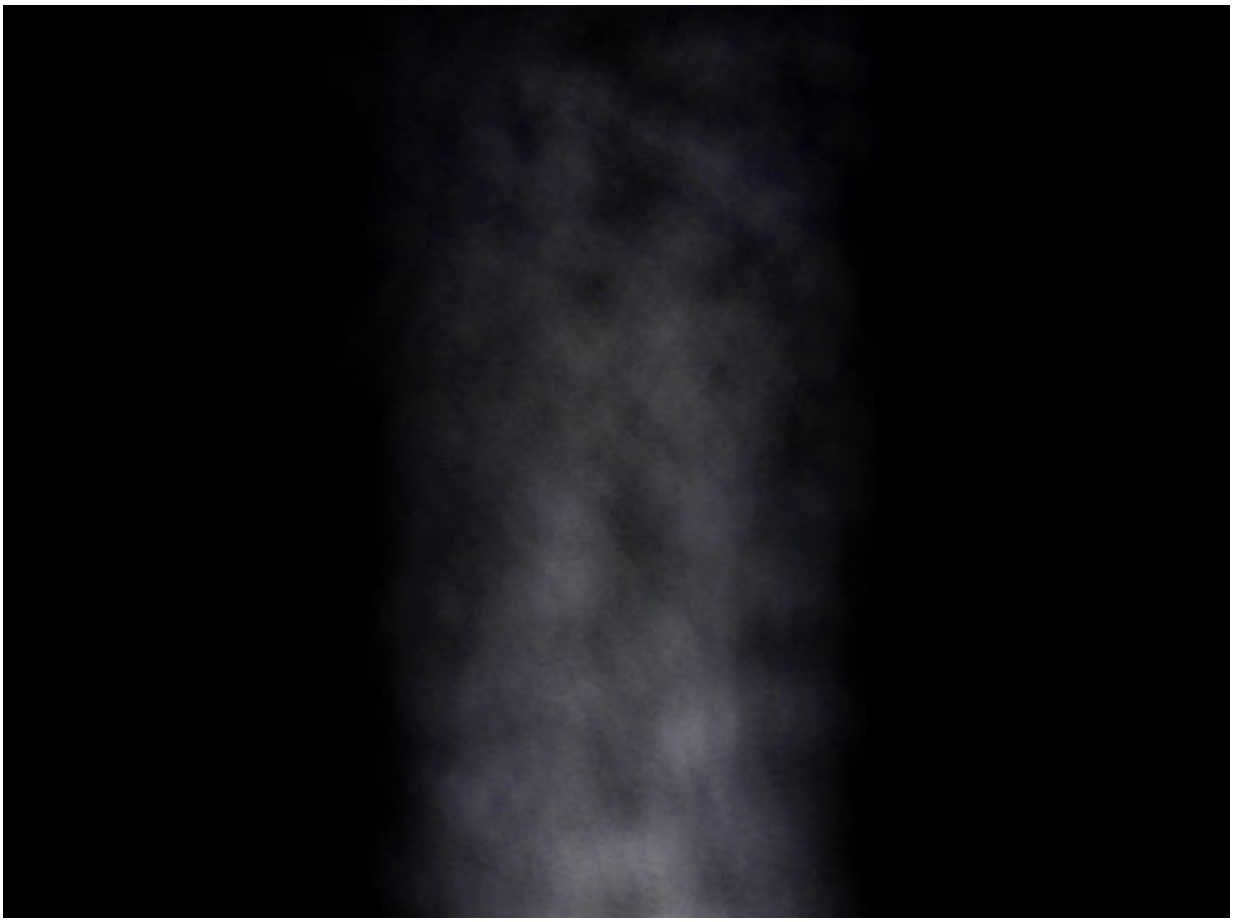
Procedural Smoke Particle System with OpenGL 2.0
Author: Tommy Hinks – tomhi761@student.liu.se
Supervisor: Stefan Gustavsson
Linköpings Tekniska Högskola, LiTH – ITN
Norrköping 05-01-24

Procedural Smoke Particle System with OpenGL 2.0



Student: Tommy Hinks – tomhi761@student.liu.se
Supervisor: Stefan Gustavsson
Linköpings Tekniska Högskola, LiTH – ITN
Norrköping 050121

Procedural Smoke Particle System with OpenGL 2.0
Author: Tommy Hinks – tomhi761@student.liu.se
Supervisor: Stefan Gustavsson
Linköpings Tekniska Högskola, LiTH – ITN
Norrköping 05-01-24

Abstract

A particle system has been implemented and with the combined use of an OpenGL application and vertex and fragment programs, smoke has been visualized. Particles where rendered using point sprites, hardware accelerated textured bill-boards. The application allows for a great deal of user control giving the user freedom to change, amongst other things, the type, color, and noise parameters of the smoke. Since the particle system is state-preserving it could be extended to include collision detection. The main bottle-neck is the number of fragments being created in the rasterization process, making small point sizes desirable.

Keywords: vertex program, OpenGL, GLSL, smoke, particle system, GPU, noise, point sprites

Procedural Smoke Particle System with OpenGL 2.0
Author: Tommy Hinks – tomhi761@student.liu.se
Supervisor: Stefan Gustavsson
Linköpings Tekniska Högskola, LiTH – ITN
Norrköping 05-01-24

Table of Contents

1 Introduction 4
2 Background Theory of a Smoke Particle System 4
2.1 Particle Movement 4
2.2 Particle Noise 5
2.3 Particle Fading 5
2.4 Life of a Particle 6
2.5 Blue edges 6
3 Implementation 6
3.1 Main Application 7
3.1.1 Point Sprites 7
3.1.2 User interface 7
3.2 Vertex Program 7
3.3 Fragment Program 8
4 Discussion 8
5 Conclusion 9
6 References 9

1 Introduction

Particle systems play an important role in computer graphics today. The particle system modeling approach is well suited for phenomena that do not rely on extreme amounts of detail; were structure and fluctuations over time are a greater concern. Gaseous phenomena are common in computer graphics and by using state-preserving particle systems these can be modelled realistically, incorporating such things as collision detection and advanced turbulence models. The aim of this project has been to model a state preserving particle system that makes extensive use of the Graphics Processing Unit (GPU). Today's graphic cards are programmable to a certain extent and some of the new features of the OpenGL 2.0 specification are programmable vertex and fragment programs. These new features put a lot of power in the hands of programmers, giving them the ability to create fast and tailor-made rendering pipe-lines. However, for reasons discussed in the section 3, the particle system does not run entirely on the GPU. Design philosophy and implementation details are described further on.

2 Background Theory of a Smoke Particle System

This section explains what characteristics have been ascribed to smoke in general and how these characteristics were modelled.

2.1 Particle Movement

The basic motion of smoke is upwards in a spiralling way. More chaotic movement is of course required and is achieved by adding 3D noise. These two components of movement are calculated separately. The simple spiralling motion is computed in the main application, the position of the particle is then sent to the vertex program where 3D noise is added to the position of the particle. The noise-function is explained in detail in section 2.2. The spiral in which the particles move is governed by the simple vector field[1]:

$$(1) \qquad \bar{F}(xpos, ypos, zpos) = \{\, 2 \cdot zpos \cdot ypos,\ yvel, -2 \cdot xpos \cdot ypos \,\}$$

where *xpos*, *ypos*, and *zpos* are the world coordinates of the particle and *yvel* is the speed at which the particles are moving upward. This value is the same everywhere in space. Gravity has a negligible effect on smoke particles and is not taken into consideration. Neither are collisions among smoke particles.

Particles are assigned velocities according their positions in the vector field. Their new positions are then calculated using first order Euler integration:

$$(2) \qquad \bar{p}(t + dt) = \bar{p}(t) + \bar{v} \cdot dt$$

where $\bar{p}(t)$ is the particle position at time *t* and $\bar{v}$ is the 3D vector representing the particle velocity, given by the vector field. As mentioned earlier this is not the final position of the particle. Noise is added to create more chaotic movement. All particles are born with a

[1] In a coordinate system where the up-vector is (0,1,0).

velocity of (0,1,0). Particles are moved according to their current velocity before being assigned a new velocity, which they keep until the next frame.

2.2 Particle Noise

The noise used on the particles differs from regular noise mainly by the value returned from it. Regular noise returns a value between zero and one, whereas the noise used for the particle system returns a 3D vector. A given point in space at a given time will yield a vector that is used to displace the point.

Every integer lattice point in space has a corresponding vector assigned to it. All these vectors are of unit length and are randomly distributed over the unit sphere. Since particles are represented as points in space and particles are moved through space in the main application the motion of the particles will look more chaotic than if they were just moved through the vector field described in equation (1).

A position in space corresponds to a position in noise space. The integer parts of the point in space are used to determine which lattice point vectors to use for interpolation. The fractional parts of the point in space are used to perform interpolation between the selected vectors to yield a single vector. Six lattice vectors are used for interpolation, two in each axis direction. The two corresponding vectors in each axis-direction are smoothly interpolated, using a C^1 continuous function, component-wise with respect to the fractional part of the point in space in that particular axis-direction. This yields three vectors that are added together and scaled according to the amplitude of the noise.

If a certain point in space always had the same noise value the smoke pillar would be static. To solve this problem the argument passed to the noise function is offset by a noise frequency multiplied by the current time. To further break up regular patterns a permutation table is used when selecting interpolation vectors.

This noise is not as mathematically exact as many other types of noise. However, observations show that since no extreme amounts of detail are required this is sufficient. The trade-off that is made in favour of speed, at the cost of nice interpolation, seems to be worth-while.

All noise calculations are done in the vertex program.

2.3 Particle Fading

Thick, warm smoke tends to be dark and transparent to begin with and grows thicker as it progresses upwards. The opposite can be said about thinner, more steam-like smoke. Both types of smoke are incorporated into the application and are separated only by a single parameter. However, once the smoke type is changed other parameters may need to be changed as well to achieve visually pleasing effects.

Fading is done using alpha values and is based on particle age. Particle age is updated and stored in the main application and sent to the vertex program using a built-in attribute

variable[2]. Alpha values are given from smooth mappings, with C^1 continuous functions, of particle age to a value between zero and one. Particle age ranges from zero to the maximum allowed age. In the case of thick, warm smoke particle age zero corresponds to an alpha value of zero, and the maximum allowed age corresponds to an alpha value of one. This means that particles become more opaque as they age. The opposite is true for steam-like smoke, were particles are opaque to begin with and fade away as they age.

2.4 Life of a Particle

All particles are born in the xz-plane. Spawn points are randomly generated upon application initialization. The spawn points are positioned by creating two circles with different radii and randomly displacing points on these circles, there is also always a spawn point at (0,0,0). The positions of these spawn points does not change during run-time. All particles are exactly identical at birth. In-between frames particle position, velocity, and age are saved.

At the beginning of each frame a fixed number of particles are born. Particles are removed when they have exceeded the maximum particle age, which is global and therefore the same for all particles. The fact that particles are born every frame results in an error at the start-up of the application. When there are few particles, two consecutive frames will be separated by a very little time-step. This leads to a dense particle cloud that rises upwards and fades away. Soon enough the application will settle at a stable frame rate and the problem no longer exists. To avoid this problem a fixed release interval for particles could be used. However, this limits the number of particles being alive at any one time, which could possibly degrade the visual quality for users on fast machines.

2.5 Blue edges

Observations have shown that most kinds smoke tend to have at least some shade of blue in them. This is modelled by adding to the blue channel of the current vertex color in the vertex program. The amount of blue added depends on the distance of the particle from the y-axis[3], the further away, the more blue is added. This feature gives good results when the smoke color is any shade of gray but may not do equally well if the user changes the smoke color so that it is dominated by one of the color channels.

3 Implementation

This section discusses implementation specific issues and explains which data structures were used. Features that are not interesting to the discussion about the particle system will be left unexplained. For those who are interested the entire source code is included in Appendix A.

The application could be run entirely on the GPU using textures as memory. The reason this has not been done is that it is beyond the scope of this project. Such an implementation requires multiple rendering targets and vertex program texture access, which is only supported by the latest graphics hardware. As of this moment these features are becoming more widely available and this sort of application is likely to become common in the future.

[2] The attribute variable used is gl_Color.
[3] City-block distance.

No lighting model is used for the particle system. This has two reasons: first of all points are used for geometry making it hard to determine normals; the second reason is that there is no obvious way in which light interacts with smoke.

3.1 Main Application

The main application was programmed in C++. Basically, what the main application does is that it stores the state of the particle system between frames and sets OpenGL state. It also takes care of initializing new particles and deleting old ones. Particles are stored in a linked list, which is traversed each frame. This data structure is purposeful since removal, insertion and traversal are simple. Particles are always inserted at the beginning of the list. This means that older particles are pushed toward the end of the list. When deleting particles all particles after the first one that is found to be too old can be deleted to, since they will be at least as old as the current one.

A class is used to handle the linked list; which uses another class that handles the nodes of the list. A node represents a particle and this is where particle information is stored. These are both small, simple classes that do not require much over-head computation due to the fact that they are classes.

The main application offers a software version of the shading calculations that are done in the vertex program. Not much work has gone into this code since it is for speed comparison only and it does not even display anything. Comparison shows that doing these calculations in software is not even half as fast as doing them on the GPU. The fact that the software version actually does fewer calculations and does not send any vertices to be rendered should be taken into consideration here.

3.1.1 Point Sprites

OpenGL 2.0 offers a convenient and fast way to render particles: hardware accelerated point sprites. The rasterization process automatically creates a textured bill-board from a single vertex; all the calculations are done in hardware. This saves doing bill-board calculations in the main application and reduces the number of vertices that have to be sent to the vertex program to one fourth. The size of the bill-boards is controlled by setting point size, which can be done either in the main application or in the vertex program.

3.1.2 User interface

All user interaction is controlled by the main application. Even the parameters that do not affect anything in the main application are stored there and sent to the vertex program as uniform variables. User input can only be made from the keyboard. For a complete list of commands see Appendix B.

3.2 Vertex Program

All the noise calculations that are described in section 2.2 are done in the vertex program. These calculations have to be done once per vertex, which is the same as once per particle. Obviously keeping the noise-calculations simple is important from a performance perspective, as particles number in the thousands.

There are a few implementational problems when creating a vertex program. As of this moment all graphics hardware does not support an infinite number of instructions in the vertex program. To make things worse different vendors support a different number of instructions[4]. This makes debugging a tedious job, especially if there is only access to one piece of graphics hardware. Arrays declared in a vertex program are not accessible procedurally; array indices have to be known at compile-time. This means that the permutation table and the lattice vectors have to be sent as uniform variables to the vertex program every time it is linked[5]. This sets limitations on the number of possible lattice vectors and the size of the permutation table. On hardware that supports texture access in vertex programs these values could be stored in textures instead.

Apart from noise, alpha fading is calculated in the vertex program. The reason it is done in the vertex program and not in the fragment program is that there are a lot more fragments than vertices, so it saves doing the computations more times.

3.3 Fragment Program

The fragment program is kept at a minimum of instructions because of the huge amount of fragments created in the rasterization process from thousands of point sprites. All the fragment program does is multiply the texture value, at all texture coordinates, by the color, which is sent as a varying variable from the vertex program. The fragment program is a major bottle-neck due to its many invocations each frame. Increasing the point size, yielding more fragments, results in performance loss. This is the reason why particles near the edges are smaller. The texture that has been used for the particles is shown in Figure 1. Areas where the checkerboard is visible are transparent.

Figure 1. Particle texture.

4 Discussion

The most obvious improvements to this application are revising the noise-function and fixing the first frames to avoid the dense cloud of particles. As suggested earlier the initial artefact could be removed by incorporating a release interval for introducing new particles, the drawbacks of this approach were also mentioned earlier. Another way of resolving this would be to start drawing particles to the screen once the frame rate has stabilized. The noise function could be improved by adding better interpolation. Removing all vector addition from the noise function and relying solely on interpolation is an interesting idea. Another major improvement to the noise-function would be to store the permutation table and the lattice vector table in textures, allowing them to be larger than they are at the moment.

It would also be interesting to add fractal noise to avoid large areas that have the same color. Introducing procedurally textured particles would also solve this issue, but would be computationally heavy. Using procedurally textured particles would allow for fewer but larger particles; this however is an entirely different approach. With procedurally textured particles

[4] See Appendix C for a list of supported hardware.
[5] Which in this particular case happens to be only once. This becomes an issue if several different vertex programs are used during rendering.

interesting effects such as glowing smoke could be introduced. This is something that is hard to implement with the approach that has been used in this project.

Probably the most interesting extension to this application would be to add some kind of collision detection. The vertex program as well as the main application would have to deal with the collisions, as it would not make sense if the vertex program moved a particle inside a solid object. This complicates matters but is in no way an unsolvable problem, at least not for simple shapes.

The fact that particle state is saved gives the possibility to have particles with different properties, such as color and maximum age. This could possibly help solve the problem of large areas having the same color.

The medium in which the smoke travels could be made more interesting by using a more elaborate turbulence function. Such effect as wind and gravity could easily be added. Particle speed could be varied according to the temperature of the medium or local vortices could be introduced. A particle system that hardly varies at all over time could be used to model clouds. In this case changing the vector field would also be necessary.

Spawn points could be moved around to create smoke trail effects. This might be used for gasoline driven machines or aeroplanes. There is no restriction that says the spawn points have to lie on a plane, any shape could contain virtually any number of spawn points.

5 Conclusion

A particle system approach is a good choice for modeling smoke. Nowadays there are a lot of options regarding whereas to do computations. Tests show that the GPU performs well in comparison to the CPU in the kind of calculations that are required. Regarding the structure of the application, as graphics hardware evolves, less and less data will be transmitted to the GPU, more and more will be done directly on the GPU. The new point sprite extension that has been used is a powerful tool for most particle systems, allowing for fast and versatile bill-boards. The combination of basic movement being controlled in the main application and noise added in the vertex program is a good combination that allows for a fast, fairly good-looking, state-preserving particle system.

6 References

Ebert D, Musgrave K, Peachey D, Perlin K, Worley S., 2003, *Texturing & Modeling.* Morgan Kaufmann Publishers, San Fransisco

Randi J. Rost, et al., 2004, *OpenGL Shading Language*, Addison-Wesley, US

Appendix A – Source Code

Main.cpp

```
/*
 * This code is based on a code skeleton for loading vertex and fragment
 * programs written by Stefan Gustavsson 2004.
 *
 * Everything that has to do with the particle system was added
 * by Tommy Hinks (thinks@hotmail.com) 2005.
*/

#include <stdio.h>     // Standard input/output
#include <io.h>        // For reading from files
#include <iostream>    // For output to console window
#include <cmath>       // Trigonometry
#include <cstdlib>
#include <time.h>      // To seed randomizer
#include <GL/glfw.h>

#include "glext.h"     // Include local glext.h
#include "ParticleList.h"         // Linked list for particles

/* Global variables for all the nice ARB extensions we need */
PFNGLCREATEPROGRAMOBJECTARBPROC  glCreateProgramObjectARB  = NULL;
PFNGLDELETEOBJECTARBPROC         glDeleteObjectARB         = NULL;
PFNGLUSEPROGRAMOBJECTARBPROC     glUseProgramObjectARB     = NULL;
PFNGLCREATESHADEROBJECTARBPROC   glCreateShaderObjectARB   = NULL;
PFNGLSHADERSOURCEARBPROC         glShaderSourceARB         = NULL;
PFNGLCOMPILESHADERARBPROC        glCompileShaderARB        = NULL;
PFNGLGETOBJECTPARAMETERIVARBPROC glGetObjectParameterivARB = NULL;
PFNGLATTACHOBJECTARBPROC         glAttachObjectARB         = NULL;
PFNGLGETINFOLOGARBPROC           glGetInfoLogARB           = NULL;
PFNGLLINKPROGRAMARBPROC          glLinkProgramARB          = NULL;
PFNGLGETUNIFORMLOCATIONARBPROC   glGetUniformLocationARB   = NULL;
PFNGLUNIFORM4FARBPROC            glUniform4fARB            = NULL;
PFNGLUNIFORM1FARBPROC            glUniform1fARB            = NULL;
PFNGLUNIFORM1IARBPROC            glUniform1iARB            = NULL;
PFNGLUNIFORM1IVARBPROC           glUniform1ivARB           = NULL;
PFNGLUNIFORM1FVARBPROC           glUniform1fvARB           = NULL;
//============================
// GL_ARB_point_parameters
//============================
PFNGLPOINTPARAMETERFARBPROC  glPointParameterfARB  = NULL;
PFNGLPOINTPARAMETERFVARBPROC glPointParameterfvARB = NULL;

//=======================================================
// Global Variables for Particle System
// (Many of which can be interactively controlled by user)
//=======================================================
GLboolean g_bShader = GL_TRUE;    // Hardware shading or not
GLfloat g_fPointSize = 43.0f;     // Particle size
float g_fTurbYVel = 1.0f;         // The speed of the smoke pillar in y-dir
GLfloat g_fMaxPartAge = 3.0f;     // Particles with age > g_fMaxPartAge are killed
GLfloat g_fAlpha = 0.028f;        // Alpha scaling

// Is the smoke warm, fast rising and grows thicker or
// is it more like steam that rises slowly and fades away
// 1.0f = steam
// 0.0f = smoke
float g_fSmokeType = 1.0f;

// Noise, default values
float g_fNoiseAmp = 0.04f;
float g_fNoiseFreq = 0.1f;

// Smoke color
GLfloat g_fSmokeColor[] = {0.5f, 0.5f, 0.5f};        // Color of smoke
```

```
// Use 3 coordinates in case we might want to spawn particles
// somewhere else than xz-plane.
#define SOURCE_POINTS 45
GLfloat g_fSourcePoints[SOURCE_POINTS][3];

// Particle vel is always up to begin with
GLfloat g_fStartingVel[] = { 0.0f, 1.0f, 0.0f };
GLboolean first_run = GL_TRUE;

// GLSL won't compile larger
// number of uniforms on GeForce FX5700...
const int TABSIZE = 32;

const GLint permTable[TABSIZE] = {
   5, 27, 9, 2, 7, 25, 22, 17, 8, 20, 16, 0, 10,
   11, 28, 3, 18, 24, 15, 6, 23, 13, 26, 31, 12, 1,
   29, 4, 21, 19, 14, 30
};

// Vectors of unit length
const GLfloat gradTable[TABSIZE*3] = {
   -0.3429, 0.8639, -0.3689,
   0.3442, 0.1373, -0.9288,
   -0.09915, 0.009929, -0.995,
   -0.3476, 0.1406, -0.927,
   0.4784, 0.6453, -0.5956,
   0.2621, 0.07418, -0.9622,
   0.2826, 0.08755, -0.9552,
   0.4016, 0.7979, 0.4496,
   0.05402, 0.9971, -0.0541,
   0.4527, 0.2878, 0.8439,
   0.1532, 0.976, 0.155,
   -0.3934, 0.8085, 0.4376,
   -0.4952, 0.431, 0.7543,
   0.2852, 0.08934, 0.9543,
   0.02181, 0.9995, 0.02181,
   -0.4823, 0.6317, 0.6069,
   0.4659, 0.6816, 0.5643,
   0.2916, 0.9061, 0.3064,
   -0.1259, 0.9839, 0.1269,
   -0.1961, 0.96, 0.2001,
   0.3097, 0.1074, -0.9448,
   -0.4533, 0.2889, -0.8433,
   0.3542, 0.1471, 0.9235,
   0.06125, 0.9962, -0.06137,
   -0.4957, 0.4345, -0.752,
   0.4675, 0.6773, 0.5681,
   0.1413, 0.02038, 0.9898,
   0.2045, 0.04372, -0.9779,
   0.1352, 0.9814, 0.1364,
   0.1559, 0.9751, 0.1579,
   0.2437, 0.9366, 0.2519,
   0.3711, 0.8351, 0.4061,
};

ParticleList *g_PartList;       // Pointer to our ParticleList

GLfloat g_fMaxPointSize = 0.0f;     // This is set in GLinit()
GLfloat g_fMinPointSize = 0.0f;     // This is set in GLinit()
//=========================================================

int running = GL_TRUE; // Main loop exits when this is set to GL_FALSE

GLuint textureID;

GLhandleARB programObj;
GLhandleARB vertexShader;
GLhandleARB fragmentShader;

// Locations in memory for uniform shader variables
GLint      location_testTexture = -1;
GLint      location_time = -1;
```

```
GLint	location_max_age = -1;
GLint	location_alpha = -1;
GLint	location_perm = -1;
GLint	location_grad = -1;
GLint	location_smoke_type = -1;
GLint	location_point_size = -1;
GLint	location_noise_amp = -1;
GLint	location_noise_freq = -1;

const char *vertexShaderStrings[1];
const char *fragmentShaderStrings[1];
GLint vertexCompiled;
GLint fragmentCompiled;
GLint shadersLinked;
char str[4096]; // For error messages from the GLSL compiler and linker

/*
 * printError() - Signal an error.
 * Prints to console window
 */
void printError(const char *errtype, const char *errmsg) {
  std::cerr << errtype << ", " << errmsg << std::endl;
}

// Print current parameter values
void printState()
{
    std::cerr << std::endl;
    std::cerr << "-----Parameters----------" << std::endl;
    std::cerr << "Point Size: " << g_fPointSize << std::endl;
    std::cerr << "Max particle age [s]: " << g_fMaxPartAge << std::endl;
    std::cerr << "Y+ velocity: " << g_fTurbYVel << std::endl;
    std::cerr << "Alpha scaling: " << g_fAlpha << std::endl;
    std::cerr << "Noise Amp: " << g_fNoiseAmp << std::endl;
    std::cerr << "Noise Freq: " << g_fNoiseFreq << std::endl;
    if(g_fSmokeType == 1)
        std::cerr << "Smoke type: steam" << std::endl;
    else
        std::cerr << "Smoke type: smoke" << std::endl;
    if(g_bShader)
        std::cerr << "Shaders: on" << std::endl;
    else
        std::cerr << "Shaders: off" << std::endl;
    std::cerr << "Color[3]: [ " << g_fSmokeColor[0] << ", " << g_fSmokeColor[1] << ", " << g_fSmokeColor[2] << " ]" << std::endl;
    std::cerr << "------------------------------" << std::endl;
    std::cerr << std::endl;
}

void printHelp()
{
    std::cerr << std::endl;
    std::cerr << "-------Help---------" << std::endl;
    std::cerr << "z/x  +/-  point size" << std::endl;
    std::cerr << "q/w  +/-  particle age" << std::endl;
    std::cerr << "y/u  +/-  Y-velocity" << std::endl;
    std::cerr << "a/s  +/-  alpha scaling" << std::endl;
    std::cerr << "j/k  +/-  Noise amplitude" << std::endl;
    std::cerr << "n/m  +/-  Noise frequency" << std::endl;
    std::cerr << "r/g/b     Cycle color channels" << std::endl;
    std::cerr << "0    reset" << std::endl;
    std::cerr << "1    shaders on/off" << std::endl;
    std::cerr << "e    smoke/steam" << std::endl;
    std::cerr << "--------------------" << std::endl;
    std::cerr << std::endl;
}

void GLFWCALL KeyFun( int key, int action )
{
  if( action == GLFW_PRESS )
   {
      switch( key )
      {
```

```
                                        // Increase/decrease particle size
                                        case 'z':
        g_fPointSize += 1.0f;
        if(g_fPointSize > g_fMaxPointSize)
        {
            g_fPointSize = g_fMaxPointSize;
        }
        printState();
        break;
                                        case 'x':
        g_fPointSize -= 1.0f;
        if(g_fPointSize < g_fMinPointSize)
        {
            g_fPointSize = g_fMinPointSize;
        }
        printState();
        break;
    // Increase/decrease y-velocity
                                        case 'y':
        g_fTurbYVel += 0.1f;
        printState();
        break;
                                        case 'u':
        g_fTurbYVel -= 0.1f;
        if(g_fTurbYVel < 0.1f)
        {
            g_fTurbYVel = 0.1f;
        }
        printState();
        break;
    // Increase/decrease alpha scaling
                    case 'a':
        g_fAlpha += 0.001f;
        printState();
        break;
                                        case 's':
        g_fAlpha -= 0.001f;
        if(g_fAlpha < 0.001f)
        {
            g_fAlpha = 0.001f;
        }
        printState();
        break;
    // Increase/decrease particle age
                    case 'q':
        g_fMaxPartAge += 0.1f;
         printState();
        break;
                                        case 'w':
        g_fMaxPartAge -= 0.1f;
        if(g_fMaxPartAge < 0.1f)
        {
            g_fMaxPartAge = 0.1f;
        }
        printState();
        break;
    // Increase/decrease noise amplitude
                    case 'j':
        g_fNoiseAmp += 0.01f;
        printState();
        break;
                                        case 'k':
        g_fNoiseAmp -= 0.01f;
        if(g_fNoiseAmp < 0.0f)
        {
            g_fNoiseAmp = 0.0f;
        }
        printState();
        break;
    // Increase/decrease noise frequency
                    case 'n':
        g_fNoiseFreq += 0.01f;
```

```
            printState();
            break;
                                        case 'm':
            g_fNoiseFreq -= 0.01f;
            if(g_fNoiseFreq < 0.0f)
            {
               g_fNoiseFreq = 0.0f;
            }
            printState();
            break;
        // Change smoke type
        case 'e':
            g_fSmokeType = (g_fSmokeType == 0.0f) ? 1.0f:0.0f;
            printState();
            break;
        // Cycle through red color channel
        case 'r':
            g_fSmokeColor[0] += 0.1f;
            if(g_fSmokeColor[0] > 1.01f)
              g_fSmokeColor[0] = 0.0f;
            printState();
            break;
        // Cycle through green color channel
        case 'g':
            g_fSmokeColor[1] += 0.1f;
            if(g_fSmokeColor[1] > 1.01f)
              g_fSmokeColor[1] = 0.0f;
            printState();
            break;
        // Cycle through blue color channel
        case 'b':
            g_fSmokeColor[2] += 0.1f;
            if(g_fSmokeColor[2] > 1.01f)
              g_fSmokeColor[2] = 0.0f;
            printState();
            break;
        case '1':
            g_bShader = (g_bShader) ? GL_FALSE:GL_TRUE;
            printState();
            break;

        // Reset button
        case '0':
            g_fPointSize = 43.0f;
            g_fTurbYVel = 1.0f;
            g_fPointSize = 50.0f;
            g_fMaxPartAge = 3.0f;
            g_fAlpha = 0.02f;
            g_fNoiseAmp = 0.04f;
            g_fNoiseFreq = 0.1f;
            g_fSmokeType = 1.0f;
            g_fSmokeColor[0] = g_fSmokeColor[1] = g_fSmokeColor[2] = 0.5f;
            printState();
            break;
        // Print help
        case 'h':
            printHelp();
            break;
        default:
           break;
        }
    }
}

/*
 * loadExtensions() - Load OpenGL extensions for anything above OpenGL
 * version 1.1. (This is a requirement from Windows, not from OpenGL.)
 */
void loadExtensions() {
    //These extension strings indicate that the OpenGL Shading Language,
    // version 1.00, and GLSL shader objects are supported.
    if(!glfwExtensionSupported("GL_ARB_shading_language_100"))
```

```
    {
        printError("GL init error", "GL_ARB_shading_language_100 extension was not found");
        return;
    }
    if(!glfwExtensionSupported("GL_ARB_shader_objects"))
    {
        printError("GL init error", "GL_ARB_shader_objects extension was not found");
        return;
    }
    else
    {
        glCreateProgramObjectARB  =
(PFNGLCREATEPROGRAMOBJECTARBPROC)wglGetProcAddress("glCreateProgramObjectARB");
        glDeleteObjectARB         = (PFNGLDELETEOBJECTARBPROC)wglGetProcAddress("glDeleteObjectARB");
        glUseProgramObjectARB     =
(PFNGLUSEPROGRAMOBJECTARBPROC)wglGetProcAddress("glUseProgramObjectARB");
        glCreateShaderObjectARB   =
(PFNGLCREATESHADEROBJECTARBPROC)wglGetProcAddress("glCreateShaderObjectARB");
        glShaderSourceARB         = (PFNGLSHADERSOURCEARBPROC)wglGetProcAddress("glShaderSourceARB");
        glCompileShaderARB        = (PFNGLCOMPILESHADERARBPROC)wglGetProcAddress("glCompileShaderARB");
        glGetObjectParameterivARB =
(PFNGLGETOBJECTPARAMETERIVARBPROC)wglGetProcAddress("glGetObjectParameterivARB");
        glAttachObjectARB         = (PFNGLATTACHOBJECTARBPROC)wglGetProcAddress("glAttachObjectARB");
        glGetInfoLogARB           = (PFNGLGETINFOLOGARBPROC)wglGetProcAddress("glGetInfoLogARB");
        glLinkProgramARB          = (PFNGLLINKPROGRAMARBPROC)wglGetProcAddress("glLinkProgramARB");
        glGetUniformLocationARB   =
(PFNGLGETUNIFORMLOCATIONARBPROC)wglGetProcAddress("glGetUniformLocationARB");
        glUniform4fARB            = (PFNGLUNIFORM4FARBPROC)wglGetProcAddress("glUniform4fARB");
        glUniform1fARB            = (PFNGLUNIFORM1FARBPROC)wglGetProcAddress("glUniform1fARB");
        glUniform1iARB            = (PFNGLUNIFORM1IARBPROC)wglGetProcAddress("glUniform1iARB");
        glUniform1ivARB           = (PFNGLUNIFORM1IVARBPROC)wglGetProcAddress("glUniform1ivARB");
        glUniform1fvARB           = (PFNGLUNIFORM1FVARBPROC)wglGetProcAddress("glUniform1fvARB");
        // Point Sprites
        glPointParameterfARB  = (PFNGLPOINTPARAMETERFARBPROC)wglGetProcAddress("glPointParameterfARB");
        glPointParameterfvARB = (PFNGLPOINTPARAMETERFVARBPROC)wglGetProcAddress("glPointParameterfvARB");

        if( !glCreateProgramObjectARB || !glDeleteObjectARB || !glUseProgramObjectARB ||
            !glCreateShaderObjectARB || !glCreateShaderObjectARB || !glCompileShaderARB ||
            !glGetObjectParameterivARB || !glAttachObjectARB || !glGetInfoLogARB ||
            !glLinkProgramARB || !glGetUniformLocationARB || !glUniform4fARB ||
                                        !glUniform1fARB || !glUniform1iARB || !glPointParameterfARB || !glPointParameterfvARB
            || !glUniform1ivARB || !glUniform1fvARB)
        {
            printError("GL init error", "One or more GL_ARB_shader_objects functions were not found");
            return;
        }
    }
}

//=========================================================
// Read and load shaders
//=========================================================
/*
 * readShaderFile(filename) - read a shader source string from a file
 */
unsigned char* readShaderFile(const char *filename) {
    FILE *file = fopen(filename, "r");
    if(file == NULL)
    {
        printError("ERROR", "Cannot open shader file!");
                                            return 0;
    }
    int bytesinfile = filelength(fileno(file));
    unsigned char *buffer = (unsigned char*)malloc(bytesinfile);
    int bytesread = fread( buffer, 1, bytesinfile, file);
    buffer[bytesread] = 0; // Terminate the string with 0
    fclose(file);

    return buffer;
}

/*
```

```
 * createShaders() - create, load, compile and link the GLSL shader objects.
 */
void createShaders() {
                        // Create the vertex shader.
    vertexShader = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);

                        unsigned char *vertexShaderAssembly = readShaderFile("vertex_shader.vert");
    vertexShaderStrings[0] = (char*)vertexShaderAssembly;
    glShaderSourceARB( vertexShader, 1, vertexShaderStrings, NULL );
    glCompileShaderARB( vertexShader);
    free((void *)vertexShaderAssembly);

    glGetObjectParameterivARB( vertexShader, GL_OBJECT_COMPILE_STATUS_ARB,
                         &vertexCompiled );
    if(vertexCompiled  == GL_FALSE)
                        {
                                        glGetInfoLogARB(vertexShader, sizeof(str), NULL, str);
                                         printError("Vertex shader compile error", str);
                        }

                        // Create the fragment shader.
    fragmentShader = glCreateShaderObjectARB( GL_FRAGMENT_SHADER_ARB );

    unsigned char *fragmentShaderAssembly = readShaderFile( "fragment_shader.frag" );
    fragmentShaderStrings[0] = (char*)fragmentShaderAssembly;
    glShaderSourceARB( fragmentShader, 1, fragmentShaderStrings, NULL );
    glCompileShaderARB( fragmentShader );
    free((void *)fragmentShaderAssembly);

    glGetObjectParameterivARB( fragmentShader, GL_OBJECT_COMPILE_STATUS_ARB,
                         &fragmentCompiled );
    if(fragmentCompiled == GL_FALSE)
                        {
                                        glGetInfoLogARB( fragmentShader, sizeof(str), NULL, str );
                                        printError("Fragment shader compile error", str);
    }
    // Create a program object and attach the two compiled shaders.
    programObj = glCreateProgramObjectARB();
    glAttachObjectARB( programObj, vertexShader );
    glAttachObjectARB( programObj, fragmentShader );

    // Link the program object and print out the info log.
    glLinkProgramARB( programObj );
    glGetObjectParameterivARB( programObj, GL_OBJECT_LINK_STATUS_ARB, &shadersLinked );

    if( shadersLinked == GL_FALSE )
                        {
                                        glGetInfoLogARB( programObj, sizeof(str), NULL, str );
                                        printError("Program object linking error", str);
                        }
                        // Locate the uniform shader variables so we can set them later:
    // a texture ID "testTexture" and a float "time".
                        location_testTexture = glGetUniformLocationARB( programObj, "testTexture" );
                        if(location_testTexture == -1)
       printError("Binding error","Failed to locate uniform variable 'testTexture'.");

                        location_time = glGetUniformLocationARB( programObj, "time" );
                        if(location_time == -1)
       printError("Binding error", "Failed to locate uniform variable 'time'.");

    location_max_age = glGetUniformLocationARB( programObj, "MaxPartAge" );
                        if(location_max_age == -1)
       printError("Binding error", "Failed to locate uniform variable 'MaxPartAge'.");

    location_alpha = glGetUniformLocationARB( programObj, "alpha" );
                        if(location_alpha == -1)
       printError("Binding error", "Failed to locate uniform variable 'alpha'.");

    location_perm = glGetUniformLocationARB( programObj, "permTable");
    if(location_perm == -1)
       printError("Binding error", "Failed to locate uniform variable 'permTable'.");
```

```
    location_grad = glGetUniformLocationARB( programObj, "gradTable");
    if(location_grad == -1)
        printError("Binding error", "Failed to locate uniform variable 'gradTable'.");

    location_noise_amp = glGetUniformLocationARB( programObj, "noiseAmp");
    if(location_noise_amp == -1)
        printError("Binding error", "Failed to locate uniform variable 'noiseAmp'.");

    location_noise_freq = glGetUniformLocationARB( programObj, "noiseFreq");
    if(location_noise_freq == -1)
        printError("Binding error", "Failed to locate uniform variable 'noiseFreq'.");

    location_smoke_type = glGetUniformLocationARB( programObj, "smokeType");
    if(location_smoke_type == -1)
        printError("Binding error", "Failed to locate uniform variable 'smokeType'.");

    location_point_size = glGetUniformLocationARB( programObj, "pointSize");
    if(location_point_size == -1)
        printError("Binding error", "Failed to locate uniform variable 'pointSize'.");
}

/*
 * setupCamera() - set up the OpenGL projection and (model)view matrices
 */
void setupCamera() {

    int width, height;

    // Get window size. It may start out different from the requested
    // size, and will change if the user resizes the window.
    glfwGetWindowSize( &width, &height );

    height = (height > 0)? height : 1;
    //if(height<=0) height=1; // Safeguard against iconified/closed window

    // Set viewport. This is the pixel rectangle we want to draw into.
    glViewport( 0, 0, width, height ); // The entire window

    glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        // 45 degrees FOV, same aspect ratio as viewport, depth range 0 to 10
        gluPerspective( 45.0f, (GLfloat)width/(GLfloat)height, 0.0f, 10.0f );

    glMatrixMode( GL_MODELVIEW );
        glLoadIdentity();
        gluLookAt( 0.0f, 2.0f, 2.0f,      // Eye position
                   0.0f, 2.0f, 0.0f,      // View point
                   0.0f, 1.0f, 0.0f );    // Up vector
}

// Update particle velocities according to vector field
void updateVel(Node *part, float dt)
{
    part->setVel(0, part->getPos(2) * part->getPos(1)*2.0f);    // Set x-vel
    part->setVel(1, g_fTurbYVel);    // Set y-vel
    part->setVel(2, -(part->getPos(0)) * part->getPos(1)*2.0f);    // Set z-vel
}

// Move, render and delete particles
void updateParticles(float time, float dt)
{
    // Insert particles....
    // Loop over source points
    for(int i = 0; i < SOURCE_POINTS; i++)
    {
        g_PartList->insert(g_fSourcePoints[i], g_fStartingVel);
    }

    // Setup rendering
    glEnable( GL_POINT_SPRITE_ARB );
    glBegin( GL_POINTS );
```

```
   // Create pointer to header node
   Node *partptr = g_PartList->getHeader();

   while(partptr->getNext() != 0)
   {
      Node *current = partptr->getNext();
      // Age the particle
      // Age first so we don't calculate unnecessary positions
      current->setAge(current->getAge() + dt);
      if(current->getAge() > g_fMaxPartAge)
      {   // If particle has aged too much, kill it
         // Also delete all nodes behind it, since
         // they will be even older...(or as old)
         g_PartList->del(partptr);
         // Break the loop, because all particles after
         // this one will also be too old...
         break;
      }
      else
      {
         // Move particle according to it's velocity
         current->setPos(current->getPos(0) + dt*current->getVel(0),
                  current->getPos(1) + dt*current->getVel(1),
                  current->getPos(2) + dt*current->getVel(2));

         // Send position and color to be rendered
         glColor4f(g_fSmokeColor[0], g_fSmokeColor[1], g_fSmokeColor[2], current->getAge());
         glVertex3fv((GLfloat *)current->getPos());

         // Update particle velocity
         updateVel(current, dt);
      }
      // Move on to next particle...
      partptr = partptr->getNext();
   }
   glEnd();
   glDisable( GL_POINT_SPRITE_ARB );
}

//======================
// Software shading
// It doesn't have the same functionality but the calculations
// are similar, though not identical, in fact, a whole lot more
// calculations are done in the shader version. These functions are for
// comparability only, speed comparability. This could well have
// been a separate program which would equally well illustrate the
// differences.
//======================

int CPUperm(int x)
{
   // Ugly abs()-hack...
   int i = int(abs(x%(TABSIZE - 1)));
   return permTable[i];
}

int CPUindex(int tx, int ty, int tz)
{
   return CPUperm(tx + CPUperm(ty + CPUperm(tz)));
}

void CPUglattice(int tx, int ty, int tz, float *out)
{
   int i = CPUindex(tx, ty, tz);
   out[0] = gradTable[i*3];
   out[1] = gradTable[i*3 + 1];
   out[2] = gradTable[i*3 + 2];
}

void CPUhnoise(float *in, float *out)
{
```

```
  // Hack to avoid negative numbers
  in[0] = in[0] + 100.0f;
  in[1] = in[1] + 100.0f;
  in[2] = in[2] + 100.0f;

  // Integer part is first decimal as integer
  // due to small scale scene.
  int ix = int(floor(10.0f*(in[0] - floor(in[0]))));
  int iy = int(floor(10.0f*(in[1] - floor(in[1]))));
  int iz = int(floor(10.0f*(in[2] - floor(in[2]))));

  // Fraction vector
  float fx = 10.0f*in[0] - ix;
  float fy = 10.0f*in[1] - iy;
  float fz = 10.0f*in[2] - iz;

  float vertices[6][3];
  // x-comp
  CPUglattice( (ix + 1), iy, iz , vertices[0]);
  CPUglattice( (ix - 1), iy, iz , vertices[1]);

  // y-comp
  CPUglattice( ix, (iy + 1), iz , vertices[2]);
  CPUglattice( ix, (iy - 1), iz , vertices[3]);

  // z-comp
  CPUglattice( ix, iy, (iz + 1) , vertices[4]);
  CPUglattice( ix, iy, (iz - 1) , vertices[5]);

  // Very simple 'interpolation', should be enough...
  out[0] = g_fNoiseAmp*(fx*vertices[0][0] + (1.0f - fx)*vertices[1][0]);
  out[1] = g_fNoiseAmp*(fy*vertices[2][1] + (1.0f - fy)*vertices[3][1]);
  out[2] = g_fNoiseAmp*(fz*vertices[4][2] + (1.0f - fz)*vertices[5][2]);
}

// Virtually the same as updateParticles()
void CPUupdatePart(float time, float dt)
{
   for(int i = 0; i < SOURCE_POINTS; i++)
   {
      g_PartList->insert(g_fSourcePoints[i], g_fStartingVel);
   }

   glPointSize(g_fPointSize);
   glEnable( GL_POINT_SPRITE_ARB );
   glBegin( GL_POINTS );
   Node *partptr = g_PartList->getHeader();

   while(partptr->getNext() != 0)
   {
      Node *current = partptr->getNext();
      current->setAge(current->getAge() + dt);
      if(current->getAge() > g_fMaxPartAge)
      {
         g_PartList->del(partptr);
         break;
      }
      else
      {
         float noise[] = { 0.0f, 0.0f, 0.0f };
         CPUhnoise(current->getPos(), noise);

         current->setPos(current->getPos(0) + dt*current->getVel(0),
                         current->getPos(1) + dt*current->getVel(1),
                         current->getPos(2) + dt*current->getVel(2));

         // We don't render anything, just observe framerate

         updateVel(current, dt);
      }
      partptr = partptr->getNext();
   }
```

```
    glEnd();
    glDisable( GL_POINT_SPRITE_ARB );
}
//======================

/*
 * renderScene() - a wrapper to drawScene() to switch shaders on and off
 */
void renderScene( float ut, float dt )
{
  float t = (float)glfwGetTime(); // Get elapsed time
                        if(g_bShader)
                        {
                          // Identify the texture to use.
                          // For some reason, this uniform variable must be set BEFORE the
                          // activation of the program object. This seems like a driver bug.
                          if( location_testTexture != -1 )
                             glUniform1iARB( location_testTexture, textureID );

                          // Use vertex and fragment shaders.
                          glUseProgramObjectARB( programObj );

      // Update the uniform time variable.
      // For some reason, this uniform variable must be set AFTER the
      // activation of the program object. This seems like a driver bug.
      if(location_time != -1 )
         glUniform1fARB( location_time, t );

      // These must be changed while the shaders are active?
      if(location_max_age != -1)
        glUniform1fARB( location_max_age, g_fMaxPartAge );
      if(location_alpha != -1)
        glUniform1fARB( location_alpha, g_fAlpha );
      if(location_noise_amp != -1)
                          glUniform1fARB( location_noise_amp, g_fNoiseAmp );
      if(location_noise_freq != -1)
                          glUniform1fARB( location_noise_freq, g_fNoiseFreq );
      if(location_point_size != -1)
         glUniform1fARB( location_point_size, g_fPointSize );
      if(location_smoke_type != -1)
         glUniform1fARB( location_smoke_type, g_fSmokeType );

      // These are never changed, so they are just set once
      // (They would have to be re-set if we re-linked shaders)
      if(first_run)
                          {
         if(location_perm != -1)
           glUniform1ivARB( location_perm, TABSIZE, permTable);
         if(location_grad != -1)
            glUniform1fvARB( location_grad, TABSIZE*3, gradTable);
         first_run = GL_FALSE;
      }

                          updateParticles(ut, dt);

                          // Deactivate the shaders.
      glUseProgramObjectARB(0);
                        }
                        else
                        {
       // Render without shaders
       CPUupdatePart(ut, dt);
                        }
}

// (A lot of the GL-state is set here and only once,
// this may be done since the primitives and rendering
// of our scene will not change much. Otherwise it is good
// programming practice to enable and disable state after
// a certain rendering.)
int GLinit()
{
```

```
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

// Culling with bill-boards, how does it work?
// Probably has no effect...
// glEnable(GL_CULL_FACE);

glGenTextures(1, &textureID);           // Generate a unique texture ID
glBindTexture(GL_TEXTURE_2D, textureID);       // Activate the texture
glfwLoadTexture2D("particle_w.tga", GLFW_BUILD_MIPMAPS_BIT); // Load image
glTexEnvi(GL_POINT_SPRITE_ARB, GL_COORD_REPLACE_ARB, GL_TRUE);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER, GL_LINEAR);

// Specify point sprite texture coordinate replacement mode
glTexEnvf( GL_POINT_SPRITE_ARB, GL_COORD_REPLACE_ARB, GL_TRUE );

glEnable(GL_TEXTURE_2D);                    // Enable texturing

// Setup blending
glEnable( GL_DEPTH_TEST );
glDepthMask( GL_FALSE );
glEnable(GL_BLEND);
glBlendFunc( GL_SRC_ALPHA, GL_ONE);

// Load the extensions for GLSL - note that this has to be done
// *after* the window has been opened, or we won't have a GL context
// to query for those extensions and connect to instances of them.
loadExtensions();
// Create the two shaders
createShaders();

int c, d;
glGetObjectParameterivARB(programObj, GL_OBJECT_ACTIVE_UNIFORMS_ARB, &c);
glGetIntegerv(GL_MAX_VERTEX_UNIFORM_COMPONENTS_ARB, &d);

std::cerr << "Vendor " << (char *)glGetString(GL_VENDOR) << std::endl;
std::cerr << "Version " << (char *)glGetString(GL_VERSION) << std::endl;
std::cerr << "Max uniforms: " << d << std::endl;
std::cerr << "Uniforms used: " << c << std::endl;
std::cerr << "Loading..." << std::endl;

glGetFloatv( GL_POINT_SIZE_MAX_ARB, &g_fMaxPointSize );
glGetFloatv( GL_POINT_SIZE_MIN_ARB, &g_fMinPointSize );
glPointParameterfARB( GL_POINT_SIZE_MAX_ARB, g_fMaxPointSize );
glPointParameterfARB( GL_POINT_SIZE_MIN_ARB, g_fMinPointSize );

// Enable vertex shader point size control
glEnable(GL_VERTEX_PROGRAM_POINT_SIZE_ARB);

// Seed randomizer
srand(time(NULL));

int count = 0;
float rand_max = (float)RAND_MAX;
// Vary radius
for(int i = 3; i > 0; i = i - 1)
{
   // Loop over angles
   for(int k = 0; k < 360; k = k + 24)
   {
      g_fSourcePoints[count][0] = (((float)(rand())/rand_max)*0.05 + (float)i/10.0f)*sin((float)k*M_PI/180.0f);
      g_fSourcePoints[count][1] = 0.0f;
      g_fSourcePoints[count][2] = (((float)(rand())/rand_max)*0.05 + (float)i/10.0f)*cos((float)k*M_PI/180.0f);
      count++;
   }
}
std::cerr << "Source Points: "<< count << std::endl;

// Create particle list.
// Has to be done after loading the extensions for some reason,
// at least when the extensions give errors.
g_PartList = new ParticleList();
```

```
    std::cerr << "Init complete" << std::endl;
    std::cerr << "Press 'h' for help!" << std::endl;
    printState();
    return 0;    // Init ok
}

int main(int argc, char *argv[])
{
                        double t,t1,t0,fps;
                        float dt;                                                   // For timing particle motion
                        float yrot = 0.0f;
                        int frames;
                        char titlestr[ 200 ];

    // Initialise GLFW
    glfwInit();

    // Open the OpenGL window
    if( !glfwOpenWindow(640, 480, 8,8,8,8, 32,0, GLFW_WINDOW) )
    {
        glfwTerminate(); // glfwOpenWindow failed, quit the program.
        delete g_PartList;
        return EXIT_FAILURE;
    }
    if( GLinit() )
    {
        glfwTerminate();
        delete g_PartList;
        return EXIT_FAILURE;
    }

    glfwSetCharCallback( KeyFun );          // Callback for keyboard

    // Do not wait for screen refresh between frames.
    // = vsync off.
    glfwSwapInterval(0);

                        frames = 0;                                                 // Framecounter
                        t0 = glfwGetTime();                                 // Initial time
                        t1 = t0;

    // Main loop
    while(running)
    {
        // Clear the color buffer and the depth buffer.
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        t = glfwGetTime();                              // Get new time
        dt = (float)(t-t1);                             // New time - Old time = How long time has passed
        t1 = t;                                         // Old time = New Time

        // Calculate and display FPS (frames per second)...
        // if a second has passed since last time this was done
        if( (t-t0) > 1.0 || frames == 0 )
        {
            fps = (double)frames / (t-t0);
            sprintf( titlestr, "Smoke [ %i particles,  %.1f FPS ]  ('h' for help)", g_PartList->getNodeCount() ,fps );
            glfwSetWindowTitle( titlestr );
            t0 = t;
            frames = 0;
        }
        frames = frames + 1;

        // Set up the camera projection.
        setupCamera();

        // Rotate scene
        yrot = (yrot > 360.0f)?(yrot = yrot - 360.0f):(yrot = yrot + 5.0f*dt);
        glRotatef(yrot, 0.0f, 1.0f, 0.0f);

        // Draw the scene.
```

```
        renderScene((float)t, dt);

        // Swap buffers, i.e. display the image and prepare for next frame.
        glfwSwapBuffers();

        // Check if the ESC key was pressed or the window was closed.
        if(glfwGetKey(GLFW_KEY_ESC) || !glfwGetWindowParam(GLFW_OPENED))
          running = GL_FALSE;
    }

    // Close the OpenGL window,terminate GLFW and free memory
    glfwTerminate();
    delete g_PartList;

    return EXIT_SUCCESS;
}
```

Node.h

```
// Author: Tommy Hinks (thinks@hotmail.com) 2005
#ifndef NODE_H
#define NODE_H

// Class defined in h-file because of it's simplicity.
// Member-functions are kept at a minimum for better performance.

class Node
{
public:
  // Constructors
  Node(float p[3], float v[3], Node *nextPtr)
  {
    pos[0] = p[0];
    pos[1] = p[1];
    pos[2] = p[2];

    vel[0] = v[0];
    vel[1] = v[1];
    vel[2] = v[2];

    age = 0.0f;
    next = nextPtr;
  }

  Node()
  {
    age = 0.0f;
    next = 0;
  }

  // ? Get ?
  // (No index checking)
  float getPos(int n) { return pos[n]; }
  float getVel(int n) { return vel[n]; }

  // Vector version, for glVertex3fv
  float* getPos() { return pos; }

  float getAge()  { return age; }
  void setAge(float t) { age = t; }

  Node* getNext() { return next; }

  // ! Set !
  void setPos(float f1, float f2, float f3)
  {
      pos[0] = f1;
      pos[1] = f2;
      pos[2] = f3;
  }
```

```
  void setVel(int n, float value)
  {
      vel[n] = value;
  }

private:
  Node* next;              // Pointer to next node in list
  float pos[3];            // Store position
  float vel[3];            // Store velocity
  float age;               // How long the particle has lived

  friend class ParticleList;
};
#endif
```

ParticleList.h

```
// Author: Tommy Hinks (thinks@hotmail.com) 2005

#ifndef PARTICLELIST_H
#define PARTICLELIST_H

#include "node.h"

class ParticleList {
public:
  ParticleList();
  ~ParticleList();

  void insert (float [3], float [3]);
  void del (Node *);
  bool empty ();
  int getNodeCount() { return nodeCount; }
  Node* getHeader() { return header; }
private:
  Node *header;
  int nodeCount;
};
#endif
```

ParticleList.cpp

```
// Author: Tommy Hinks (thinks@hotmail.com) 2005

#include "ParticleList.h"

ParticleList::ParticleList ()
{
   header = new Node();    // Initialize header
   nodeCount = 0;
}

ParticleList::~ParticleList()
{
   if ( !empty() )
   {
      Node *currentPtr = header;
      Node *tempPtr;

      while ( currentPtr != 0 )
      {
            tempPtr = currentPtr;
            currentPtr = currentPtr->next;
            delete tempPtr;
      }
   }
}

// Contents of node as argument
```

```
void ParticleList::insert(float p[3], float v[3])
{
    // Insert node, directly after header node
    header->next = new Node(p, v, header->next);
    nodeCount = nodeCount + 1;
}

// Delete the node 'in front' of this one
// Also means 'header' can never be deleted...
// (since no node points to header)
void ParticleList::del(Node *current)
{
   Node *tempptr;
   Node *delptr = current->next;

   while(delptr !=0)
   {
      tempptr = delptr->next;
      delete delptr;              // Free memory
      delptr = tempptr;
      nodeCount = nodeCount - 1;   // Decrease count
   }

   // The list now ends after the node we sent in
   current->next = 0;
}

bool ParticleList::empty ()      // Check if list is empty
{
   return (header->next == 0);
}
```

vertex_shader.vert

```
// Author: Tommy Hinks (thinks@hotmail.com) 2005

uniform float time;
uniform float MaxPartAge;   // Maximum particle age
uniform float alpha;        // Alpha scaling
uniform float pointSize;    // Point size
uniform float smokeType;

// Noise
uniform float noiseAmp;
uniform float noiseFreq;

// Better to do color calculations in vertex shader since
// there are less vertices than fragments.
// Constants for blue fade
const float blueness = 0.2;
const float blue_fade = 1.0;

const int TABSIZE = 32;     // Permutation stuff
const float TABMASK = 31.0;
//const float OCTAVES = 4.0;  // For fractal "noise"

uniform int permTable[TABSIZE];
uniform float gradTable[TABSIZE*3];

varying vec4 v_color;

int perm(int x)
{
   return permTable[int(mod(float(x), TABMASK))];
}

int index(int tx, int ty, int tz)
{
   return perm(tx + perm(ty + perm(tz)));
}
```

```
vec3 glattice(int tx, int ty, int tz)
{
    int i = index(tx, ty, tz);
    return vec3(gradTable[i*3], gradTable[i*3 + 1], gradTable[i*3 + 2]);
}

vec3 hnoise(vec3 invec)
{
   // Hack to avoid negative numbers
   invec = invec + vec3(100.0, 100.0, 100.0);

   // Integer part is first decimal as integer
   // due to small scale scene.
   vec3 ipart = floor(10.0*fract(invec));
   int ix = int(ipart[0]);
   int iy = int(ipart[1]);
   int iz = int(ipart[2]);

   // Fraction vector
   vec3 fracvec = 10.0*invec - ipart;

   vec3 vertices[6];
   // x-comp
   vertices[0] = glattice( (ix + 1), iy, iz );
   vertices[1] = glattice( (ix - 1), iy, iz );

   // y-comp
   vertices[2] = glattice( ix, (iy + 1), iz );
   vertices[3] = glattice( ix, (iy - 1), iz );

   // z-comp
   vertices[4] = glattice( ix, iy, (iz + 1) );
   vertices[5] = glattice( ix, iy, (iz - 1) );

   vec3 result;

   // Simple 'interpolation', should be enough...
   result = vertices[1] + smoothstep(0.0, 1.0, fracvec.x)*(vertices[0] - vertices[1]) + vertices[3] + smoothstep(0.0, 1.0,
fracvec.y)*(vertices[2] - vertices[3]) + vertices[5] + smoothstep(0.0, 1.0, fracvec.z)*(vertices[4] - vertices[5]);

   return result;
}

void main( void )
{
    // NOT USED!!
    /*
    vec3 vHnoise;
    float i;
    float freq = time*noiseFreq;

    // Fractal "noise"
    // Due to the fact that noiseAmp takes on small values
    // the fractal "noise" may be indistinguishable(?) from just "noise"
    for(i = 0.0; i < OCTAVES; i++)
    {
        // (Powers of two could be precalculated and stored in
        // a texture or a uniform array to speed up program, and so could
        // their inverses...)
        float exp2 = exp(i);
        vHnoise = vHnoise + (1.0/exp2)*hnoise(gl_Vertex.xyz - exp2*freq);
    }
    */

    vec3 vHnoise = hnoise(gl_Vertex.xyz - time*noiseFreq);

    // Add "noise" to gl_Vertex
    vec3 glVert = vec3(gl_Vertex.xyz) + noiseAmp*vHnoise;
    gl_Position = gl_ModelViewProjectionMatrix * vec4(glVert, 1.0);

    v_color = gl_Color;
```

```
    // Calculate alpha fade
    v_color.a = alpha*(smoothstep(smokeType*MaxPartAge, (1.0 - smokeType)*MaxPartAge, v_color.a));

    // Use city block distance for blue edges...
    v_color.b = v_color.b + blueness*smoothstep(0.0, blue_fade, (abs(glVert.x) + abs(glVert.z)));

    // Set pointsize
    // Smaller points at the edges enables smaller pointSize,
    // which reduces total number of fragments and speeds up the program.
    gl_PointSize = min(pointSize - 10.0*(abs(glVert.x) + abs(glVert.z)), gl_Point.sizeMax);
}
```

fragment_shader.frag

```
// Author: Tommy Hinks (thinks@hotmail.com) 2005

uniform sampler2D testTexture;

// Fragment color
varying vec4 v_color;

void main( void )
{
    // Multiply color and texture values
    gl_FragColor = v_color*texture2D( testTexture, gl_TexCoord[0].st );
}
```

Appendix B – User interaction

These are available while running the program by pressing 'h'.
The list of commands will appear in the console window.

z/x	+/-	point size
q/w	+/-	particle age
y/u	+/-	Y-velocity
a/s	+/-	alpha scaling
j/k	+/-	Noise amplitude
n/m	+/-	Noise frequency
r/g/b		Cycle color channels
0		reset
1		shaders on/off
e		smoke/steam

Appendix C – Supported Hardware

The program has known compatibility issues. It does not seem to run at all on ATI graphic cards. The following lists present hardware that it has and has not run on:

Working

nvidia GeForce FX5900
nvidia GeForce FX5700
nvidia GeForce4 4200 Go (laptop)

Not Working

ATI Radeon 9800
ATI Radeon 9700 pro
ATI Radeon 9600