Univerzita Komenskeho v Bratislave
Fakulta Matematiky, Fyziky a Informatiky

Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky

OpenGL knižnica pre časticové systémy

Bakalárska práca

2012

Michal Bubnár

Univerzita Komenskeho v Bratislave
Fakulta Matematiky, Fyziky a Informatiky

Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky

OpenGL knižnica pre časticové systémy

Bakalárska práca

Študijný program: Informatika

Študijný odbor: 2508 Informatika

Školiace pracovisko: Katedra Informatiky

Školiteľ: RNDr. Martin Samuelčík

Bratislava, 2012

Michal Bubnár

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Michal Bubnár
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: OpenGL knižnica pre časticové systémy

Cieľ: Cieľom práce je naprogramovanie C, C++ knižnice na definovanie a efektívne zobrazovanie časticových systémov. Aktualizácia systému v čase sa bude uskutočňovať pomocou OpenGL rozšírenia EXT_transform_feedback. Podobne vykreslenie bude čo najefektívnejšie s pomocou najnovších prostriedkov OpenGL.

Vedúci: Mgr. Martin Samuelčík, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky

Dátum zadania: 05.10.2011

Dátum schválenia: 26.10.2011 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Týmto sa chcem poďakovať vedúcemu práce RNDr. Martinovi Samuelčíkovi za podporu a poskytnutie odborných konzultácií.

Abstrakt

Autor:	Michal Bubnár
Nazov bakalárskej práce:	OpenGL knižnica pre časticové systémy
Škola:	Univerzita Komenskeho v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Veduci bakalárskej prace:	RNDr. Martin Samuelčík
Rozsah práce:	49 strán

Bratislava, jún 2012

V práci sa zaoberáme časticovými systémami, rozoberáme ich vlastnosti a pokusíme sa všetky poznatky implementovať viacerými spôsobmi, ktoré následne porovnáme, aby sme zistili, ktoré sú najvhodnejšie a najrýchlejšie. Z technickej stránky používame otvorený štandard OpenGL, presnejšie jeho verziu 3.3, ktorá obsahuje pokročilú funkcionalitu na prácu s grafickou kartou a grafickým procesorom, vďaka čomu vieme efektívne implementovať časticový systém a rýchlo vykresliť a spracovať veľké množstvá častíc. Na základe týchto poznatkov vytvoríme knižnicu na vykresľovanie časticových systémov a API, pomocou ktorého sa časticové systémy budú dať ľahko integrovať do aplikácie koncového uživateľa.

Kľúčové slová: Časticový systém, OpenGL, GPU, Shader, Transform Feedback, Rendering

Abstract

Author:	Michal Bubnár
Thesis title:	OpenGL library for particle systems
University:	Univerzita Komenského v Bratislave
Faculty:	Faculty of mathematics, physics and informatics
Department:	Deparment of informatics
Advisor:	RNDr. Martin Samuelčík
Thesis length:	49 pages

Bratislava, june 2012

In this work we deal with particle system, we examine their properties and then we will try to implement the knowledge with multiple techniques, that will be compared in order to find out, which ones are the best and fastest. From technical point of view we use open standard OpenGL, more precisely version 3.3, that includes advanced functionality to work with graphics card and graphical processing unit. With it, we can effectively implement particle system and quickly render and update big amounts of particles. Based on this knowledge we will create a library for rendering particles and API, that will deploy particle systems in end-user application easily.

Key words: Particle System, OpenGL, GPU, Shader, Transform Feedback, Rendering

Contents

List of abbreviations

OpenGL - Open Graphics Library

GPU - Graphics Processing Unit

CPU - Central Processing Unit

GLSL - OpenGL Shading Language

VAO - Vertex Array Object

VBO - Vertex Buffer Object

glm - OpenGL Mathematics Library

API - Application Programming Interface

FPS - Frames Per Second

Introduction

Particle systems play important role in modern graphics and computer games. They are used to model effects, which natural behavior is chaotic, irregular and hard to model using math functions, because of their irregularity and complexity. To such effects belong fire, explosions, electric sparks etc. Their usage is not only this, they are also used for simulating fluids ad several others physically based simulations. All these effects have one thing in common - in order to simulate them, we must work with thousands of small particles, update them and render them to achieve desired effect.

The aim of our this work is to create usable and portable C++ library, that will handle particle systems creation, updating and rendering. It must be as flexible as possible, so that user of this library can use it to fit any of his needs. And at top of all, it must be as effective and fast as possible. To achieve this, we will use the latest graphics programming paradigms, so that we can utilize GPU computational power to deal with particle updating and rendering. This library will also be written in platform-independent manner, it won't use any system-specific calls. However, demo applications will be written under Windows. We'll use OpenGL for rendering, because it's cross-platform, free, and it is supported by most graphics card vendors.

For users not having graphic card that supports some of the latest OpenGL extensions, there are also simpler implementations of particle system in this library, that run mainly on CPU. We will also do a comparison of different implementations of particle systems to see which one performs best.

Together with this library, we will also prepare additional library with some basic effects, so that user can create them as straightforwardly as possible.

This work builds on Transform Feedback Particle System tutorial, which is available at [TFtut] and extends it to the level where user can create arbitrary number of particle generators with specific settings.

Chapter 1

OpenGL

To actually implement our particle system library, we will use OpenGL for rendering and updating particles. OpenGL is open, standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics. OpenGL was developed by Silicon Graphics Inc. (SGI) in 1992, and today is managed by the non-profit technology consortium Khronos Group ([Khronos]). It is widely used in industry, as well as video games for data visualization.

It is a common mistake to think that OpenGL is a library for computer graphics. Even though we use it like that, OpenGL itself is just a specification - a set of functions along with their associated parameters and constants, through which we can produce graphics. And it is up to graphic card vendors to actually implement the specification in the their products while taking advantage of their hardware to achieve maximum performance. All major graphic card vendors, like AMD and nVidia support OpenGL to their latest versions (currently it's 4.2), and almost all minor vendors implemented at least part of OpenGL specification into their graphic cards, so we can be sure that our implementation of particle system will run almost anywhere. And because OpenGL is supported on all major operating systems (Windows, Mac, Linux), our library will be cross-platform. There are also software implementations of OpenGL specification (Mesa3D is the most known), but they're much slower than hardware implementations.

The programmer usually creates a system dependent window to render into, then creates OpenGL rendering context within that window in order to be able to call OpenGL functions. This approach makes porting OpenGL applications along platforms very convenient.

1.1 Evolution of OpenGL

OpenGL has gone a long way, since its first introduction in 1991. Prior to OpenGL version 2.0 (further referred as old OpenGL), most of the its functionality was fixed. This basically means, that only features, that are implemented in according to specification into the graphics hardware, could be used. OpenGL back then was practically fixed-function state machine with built-in variables, such as projection and transformation matrices, that we could freely read and modify. But because GPUs started to become more and more flexible and programmable, so must have done OpenGL. This eventuated into OpenGL as an interface with complex data-flow, that includes several application-programmable stages. Not only that, the performance of OpenGL has increased in many important raw graphics operations.

Today, OpenGL is a modern API, being maintained by Khronos Group, a non-profit organization that is focused on creation of open standards and free APIs. Its members are leaders and innovators in computer graphics field, employees of leading companies like nVidia and AMD and many other experts in the field of 3D graphics. Another advantage is, that OpenGL has many variants for other devices than just computers, like OpenGL ES (embedded systems), so it shouldn't be a problem to port the particle system library to mobile devices with operating systems that support OpenGL ES, like Android or iPhone.

1.2 OpenGL Shading Language

This is the most important feature in modern hardware era. OpenGL Shading Language (further referred as GLSL) is a language for writing shaders, allowing us to execute our custom per-vertex (along with additional geometry creation) and per-fragment operations directly on GPU. It's derived from C language and resembles C++ in many ways. GLSL has many built-in variable types and functions, that are commonly used in 3D graphics, for example matrices - there's a different variable type for matrix dimensions up to 4x4, vertices - there are 2D, 3D and 4D vectors available. The functions that are fundamental for 3D graphics are also available. To those belong functions like dot product, cross product, smooth interpolation between values, all goniometric functions and much more. In this work, I'll be using only GLSL version 3.30, that comes along with OpenGL 3.3. The whole language specification can be found at [GLSLref].

1.2.1 OpenGL Rendering pipeline

The whole process of transforming input data into final 2D image is quite complex, but one can imagine it as a rendering pipeline, where at the entrance there are definitions of vertices and their attributes along with polygons that they form, texture data and other resources needed for rendering and on the end there's a final 2D image, that is displayed on the screen. Older OpenGL provided only fixed vertex and fragment processing. A simplified rendering pipeline is shown in Figure 1.1:

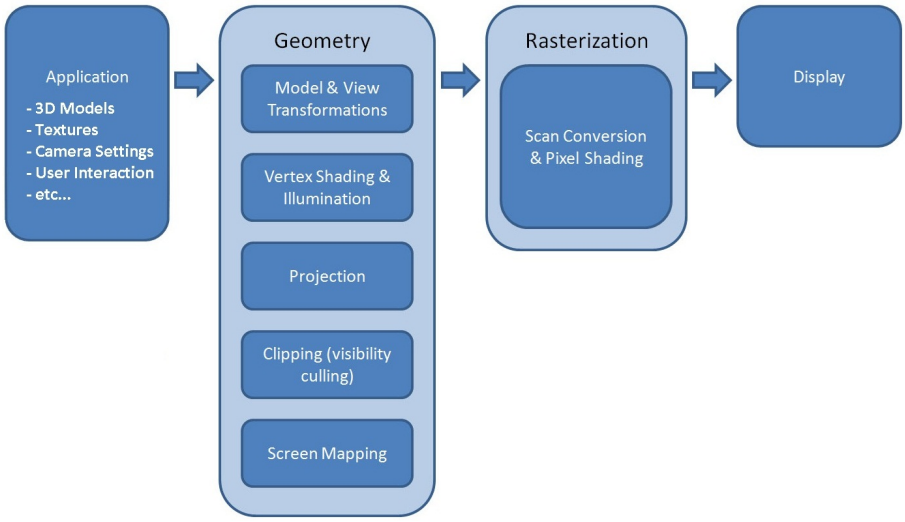

Figure 1.1: OpenGL rendering pipeline

Vertex processing is usually responsible for transforming vertices into window coordinates, but it can calculate and output additional vertex info, such as its position in eye-space coordinates.

After polygon rasterization, fragment processing takes place. This is done by fragment shader, which is executed on every single pixel, or fragment. Here OpenGL deals with hidden surface removal using the depth buffer. This means, that everytime a new polygon comes, every pixel it should occupy has its depth calculated, and this depth is checked against the values already in depth buffer. Depending on depth test properties, only values that meet certain criteria (but basically lower values are taken, which means less depth and thus closer to us) will pass and depth buffer is updated. Only fragments, that passed depth test will qualify for further processing, which usually includes applying right color depending on material properties, texture used, lighting parameters etc.

The Geometry field and Rasterization field can be replaced by our own programs now. Starting from OpenGL 3.3, where all fixed functionality is removed, it's obligatory and we must program

these parts of rendering pipeline. Even though it may not seem so at first glance, this approach has many advantages for us. For example, there may not be always need for matrices, if we are only to render simple static 2D graphics, we can write a shader that will take data directly in window coordinates, and output them right away. This way we can save some processing power. We can customize the rendering pipeline to exactly fit our needs. In this work, GLSL version 3.3 will be used.

1.2.2 Shader types

There are currently 3 basic shader types - **vertex** shaders, **fragment** shaders and **geometry** shaders. In this work. We will shortly describe each one of them:

- **Vertex Shader** - This shader processes input vertices and their associated attributes, and passes results further into OpenGL rendering pipeline (geometry shader, if it's present)
- **Geometry Shader** - In this shader we can group vertices and form triangles from them, or we can discard them to save some processing power. This type of shader is used to tesselate complex mathematical shapes for example, but can be used for more general purposes. It will be very important in this work.
- **Fragment Shader** - Takes care of producing final image, gets called for each pixel (fragment) and is used for creating lighting effects or interpolating colors among vertices.

1.2.3 Shader programs

Single shader itself isn't enough to replace pipeline parts - in order to make it work, we must compile one or more shaders and then link them together to create a **Shader program**. One can think of a shader as a single source code file. And when compiling more source files together, the linkage must be done. Additionally, every shader type - vertex, fragment or geometry must have a main method, just like in C or C++, even though can be comprised of multiple source files (we can just have a file with some custom function definitions which are used in multiple shader programs). After we have successfully linked a program, we can finally use it. We can have several shader programs, as they can serve different purposes, and then quickly switch between them in our application.

Starting from OpenGL 3.1, we don't set vertex positions, colors or its texture coordinates. They're are referred to as vertex attributes and it is upon us to decide which attribute will represent position, color and so on. Each of vertex attributes has an index, so we can assign position to index 0, texture coordinates to index 1 and so on. We setup all these generic vertex attributes before rendering. There's a keyword `layout` in GLSL to assign input variables with vertex attributes coming to shader. Here is an example of a very simple shader program, consisting of one vertex shader, that just sends incoming vertex positions further into rendering pipeline, geometry shader that creates a triangle from every 3 vertices passed and one fragment shader, that produces red fragments.

```
#version 330

in vec3 inPosition;

void main(void)
{
  gl_Position = vec4(inPosition, 1.0);
}
```

Listing 1.1: Trivial vertex shader

```
#version 330

layout(triangles) in;
layout(triangles) out;
layout(max_vertices = 3) out;

void main()
{
  gl_Position = gl_in[0].gl_Position;
  EmitVertex();
  gl_Position = gl_in[1].gl_Position;
  EmitVertex();
  gl_Position = gl_in[2].gl_Position;
  EmitVertex();

  EndPrimitive();
}
```

Listing 1.2: Trivial geometry shader

```
#version 330

out vec4 outColor;

void main(void)
{
  outColor = vec4(1.0, 0.0, 0.0, 0.0);
}
```

Listing 1.3: Trivial fragment shader

1.2.4 Variable modifiers

Usually, we want the subsequent shaders in pipeline to communicate with one another, especially vertex and fragment shader often share many data. This can be done using variable modifiers, that GLSL has built-in. If we want a variable from one rendering stage to be passed to another, we do it using `in` and `out` variable modifiers. To ensure the consistency, a variable that has an `out` modifier in one stage must have the `in` modifier and same name in another pipeline stage. There are also special qualifiers for communication between vertex and fragment shader, to resolve ambiguity. Because vertex shaders are executed once per vertex and fragment shaders once per fragment, we need to feed the fragment shader data not only in the vertices, but also among the whole rasterized polygon. There are several modifiers to do that. If we declare output variable from vertex shader with `smooth` keyword, then in the subsequent stage of rendering pipeline their values will be interpolated throughout all fragments that the polygon covers. It means, that the

closer particular fragment to source vertex is, the more from its value is present in the fragment. This is useful for interpolating colors, but also normals to achieve per-pixel lighting. Another possible modifier is `flat`, that doesn't perform interpolation, but takes the value from a single vertex instead and puts it all over rasterized polygon. The vertex that acts as a source of this data is called *Provoking Vertex*. It's usually either first or last vertex of polygon.

There are situations, in which we need to communicate with shader program from our main program. GLSL has built mechanism for doing exactly this. There is a variable modifier `uniform`, that allows us to change its value from within our program. The word uniform represents the variable's attribute characteristic - that its value is uniform - i.e. it doesn't change during execution of vertex program. We must set its value before calling any rendering function, and then throughout rendering, its value remains the same. The example usage of this variable modifier is for setting modelview and projection matrices from within our program to transform our vertices properly. Uniform variables can't be changed from within the shader, they're read-only there. Listings 1.4 and 1.5 are an example of a shader program, that transforms vertices using modelview and projection matrix, and smoothly interpolates colors throughout whole polygon.

```
#version 330

layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inColor;

smooth out vec3 theColor;

void main()
{
  gl_Position = vec4(inPosition, 1.0);
  theColor = inColor;
}
```

Listing 1.4: Color vertex shader

```
#version 330

smooth in vec3 theColor;
out vec4 outputColor;

void main()
{
  outputColor = vec4(theColor, 1.0);
}
```

Listing 1.5: Color fragment shader

In our work, we'll rely heavily on shaders, as most particle system computations will be performed on GPU and will be implemented in shaders. For the GLSL manual and specification, refer to [GLSLref].

1.3 Geometry rendering

Rendering geometry in OpenGL now is different than rendering back then before shaders. The early versions of OpenGL used so-called immediate rendering mode, in which user explicitly declared vertices and their attributes using OpenGL functions like `glVertex` or `glTexCoord`. The sample triangle render with a texture applied onto it is shown in Listing 1.6.

```
glBegin(GL_TRIANGLES);
  glTexCoord2f(0.0f, 0.0f); glVertex2f(-10.0f, 0.0f);
  glTexCoord2f(1.0f, 0.0f); glVertex2f(10.0f, 0.0f);
  glTexCoord2f(0.0f, 0.5f); glVertex2f(5.0f, 5.0f);
glEnd();
```

Listing 1.6: Triangle in immediate render mode

But as the GPUs were evolving, this approach turned out to be slow because the constant flow of identical data from CPU to GPU was causing a bottleneck on CPU - GPU had to wait for CPU to send data before it could perform rendering. Because of that, OpenGL 1.5 introduced Vertex Buffer Objects (VBOs) - the data storage buffers residing in GPU memory instead of client memory. This allows us to upload arbitrary data to GPU, and then, instead of rendering using immediate mode, we just have to tell OpenGL the way data is stored in the VBO (for example byte offsets between two consecutive vertices, order of attributes etc.) and call one drawing function, like `glDrawArrays`. Prior to OpenGL 3.0, vertex attributes were set using functions as `glVertexPointer`, `glTexCoordPointer`, or `glNormalPointer`. However, these functions were deprecated and removed from OpenGL 3.1 core, while replaced with a more general Vertex

Array Objects.

1.3.1 Vertex Array Objects

Vertex Array Object (VAO) is OpenGL state object, adopted to core OpenGL in version 3.0. It stores bindings between vertex attributes and their associated VBOs. Vertex attributes are completely arbitrary and they're indexed with numbers. It is up to us to decide which index represents position, texture coordinate, normal, color or any other attribute that we come up with. These attributes come as `in` variables into vertex shader. To set concrete vertex attribute, function `glVertexAttribPointer` is used. With it, we set attributes index, data type (float, integer etc.), offset between two consecutive attributes and byte offset from start of VBO, where first occurence of this attribute is located. For detailed explanation and usage, see [VertArray]. There are also additional buffer object types in OpenGL, though we don't need them in this work - pixel buffer, texture buffer, framebuffer, renderbuffer, uniform buffer transform feedback buffer and some other. See [GLregistry] for more.

1.4 GLM

GLM is a library for OpenGL mathematics. Because all fixed funcionality was removed, we cannot use `glu` (OpenGL utility library) functions anymore to setup basic things like perspective projection. This library is free to use and is a decent descendant of `glu`. It has all fundamental data types and functions when working with 3D graphics - vertices and matrices and all common operations with them. In addition, it has all function alternatives that were in fixed functionality, like `glTranslate`, `glRotate`, `gluLookAt` etc. We will use this library in our work very often. For documentation and example usage, refer to [GLM].

Chapter 2

Particle Systems

2.1 Particle System

Particle system is a technique to simulate certain effects or phenomena, that are otherwise very difficult to reproduce using conventional rendering techniques. Examples of such effect are fire, sparks, dust, moving water, smoke, explosions etc. All these objects have something in common - they do not have smooth, well-defined surfaces; instead they are very complex and irregular. The idea to simulate these "fuzzy" phenomena is to create many small particles, that act as independent units, and then render them together to create desired effect.

Typically, particle system's position and motion are controlled in 3D space by what is reffered to as emmiter, or generator. It acts as a source of particles, and it's position in 3D space determines the position of particles generated by it. Furthermore, a certain set of parameters is set to an emitter. These parameters usually include initial particle velocities, or directions they will be moving to, initial color or texture, initial lifetimes and spawning rate, or how many particles to generate. To make the whole thing irregular and "fuzzy", instead of defining a precise value for newly generated particles, a particle system designer usually specifies a central value and some degree of randomness on either side of the center. For example the lifetime of a new particle may be 1 second +- 0.2 seconds. With uniform random distribution, the average lifetime of particle is thus 1 second. And if apply this randomness to all parameters (color, velocities), the final effect has an irregular, hard to define behavior, which is actually what we want. For more detailed overview and general information about particle systems, see [REE83] and [REY87].

2.2 Basic Model Of Particle System

In each frame where particle system is used, 4 basic steps are performed:

1. Generation of new particles - we create some particles and set their initial parameters
2. Rendering - particles are rendered to framebuffer
3. Updating living particles - every particle that is still "alive", i.e. still has a lifetime is moved and transformed according to its dynamic properties
4. Removing old particles - every particle that has an expired lifetime is removed from system

2.2.1 Generation of new particles

Each new frame, new particles are added to a system. The number of particles is important, because it strongly influences density of the fuzzy object that is being modeled. It is up to particle system designer to determine the optimal number of particles to achieve particular effect. If we set the number of particles, let's say N, with a variance v (so that actual number of new particles is in interval $< N - v, N + v >$), then every frame (or specified time interval) the count of added particles is:

$$N_{added} = \frac{(N+rand()*V)*t_{elapsed}}{t_{int}}$$

Where $rand()$ is a function returning value from interval $< -1.0, 1.0 >$ and $t_{elapsed}$ is the time elapsed since previous frame. If the time between two frames or number of particles is too low, then it's possible that N_{added} will be less than 1, which is undesired, because number of particles added must be integer. To avoid this, we can use a floating point variable for N_{added} and whenever this variable exceeds 1.0, we convert it to integer and add that many particles.

Now after the number of generated particles is known, the particle system must determine the values for their attributes. The most common attributes are particle position, velocity, size, color, transparency, or lifetime. These attributes can be ranging from very simple to very complex - depending on the complexity of the phenomenon.

There are also several parameters of particle system that control initial position of generated particles. A Generator has a position in 3D that defines its origin, and a generation shape, which defines the region around origin into which newly generated particles can be placed, for example spherical region. The number of control attributes of particle system is limitless and it's up to a designer to choose those which will fit his model most.

2.2.2 Rendering

In this phase frame data are generated according to particles data. This phase mostly affects overall performance of particle system and thus it is crucial to implement this phase as effective as possible. The rendering of particles is generally equally computationally expensive as rendering of polygons, because particle can overlap, be transparent and also cast shadows on other particles. Furthermore, particles can intersect objects modeled by primitives in our scene. With this in mind, there is no point in trying to simulate all these things, because of great computational expense. For these reasons, we will assume, that we don't need to write depth values of particles against to framebuffer, only test them against it. Also, we will consider each particle to be a point light, so that every pixel the particle will have its intensity increased, depending on color and transparency of particle.

2.2.3 Updating living particles

Each frame, all living particles are updated. Their velocity is added to their position, lifetime is decreased and also color may change. To add more complexity, particle system designer may also add an acceleration factor as parameter of particle system to alter velocity of each particle from frame to frame. This allows to simulate effects like gravity and make particles move not only along lines, but also parabolic and other curves.

2.2.4 Removing old particles

Upon generation, particle is given certain lifetime. This lifetime is decreased every frame, and when its below a certain threshold (usually zero), particle is destroyed. Generally, we either remove particle data from memory, or set its flag as dead, so that this particle can be reinitialized in subsequent frames. The second approach is better, because actual removal of particle from memory can have a performance penalty.

2.3 Generating particles

Particle system implemented in this work will consist of multiple generators with different properties, that emit particles. Whenever generator is transformed, so are all of its descendant particles. All properties of particles are also set according to their parent generator settings. Also the number of new particles generated by particle system is dependent on parent generator generation rate.

There are already several solutions implementing particle systems, such as Flint ([FLINT]) for Adobe Flash or SPARK for both DirectX and OpenGL ([SPARK]).

Chapter 3

Implementation

In this chapter, we will discuss the way library is implemented. We'll go through most important parts of code and explain them. We'll also go through the API and explain what is each function used for.

Library itself consists of 2 parts - one is a relatively low level API for creating particles and particles generators through which user can create variety of effects to fit any of his needs. The second one is a collection of preprogrammed particle effects, which are commonly used anywhere, such as fire, explosion, smoke etc. Even though these effects have only several parameters to control and thus they are not as flexible as the low level API, they are quite handy and may reduce the code overhead for simple applications, where these pre-programmed effects are sufficient.

The name of particle system library is Blaze Particle System Library. It is available at website [BLAZE] under open source license.

3.1 Library types

In general, there are two library types: those that link statically and those that link dynamically. We will explain these terms in few words, and will talk about advantages and disadvantages of both:

3.1.1 Static linking

Static linking is the method used to combine the library routines with the application in a way, that when program is being built, the linker will search all the libraries for the unresolved functions

called in the program and if found, it will copy the corresponding object code from the library and append it to the executable. The linker will keep searching until there are no more unresolved function references. This is the most common method for linking libraries. This method boasts of ease of implementation and speed but will generate bulky executable. However, final executable isn't now dependent on any other files, because all used parts have been copied into it.

3.1.2 Dynamic linking

In dynamic linking, the required functions are compiled and stored in a library. On Windows systems, such libraries have .dll extensions, which stands for Dynamically Linked Library. Unlike static linking, no object code is copied in to the executable from the libraries. Instead of that, the executable will keep the name of the library in which the required function resides. And when the executable is running, it will load the library and call the required functions from it. This method yields an executable with small footprint, but have to compromise on speed a little, but on modern systems, we can say that this tie is negligible. The main advantage of this method is the reusability of code. If the same library is used within multiple applications, the space savings become clearly visible. Not only that, if there is a new version of particular library, applications using the ones statically linked must be recompiled in order for new version to take effect, whereas in dynamic linking, we only replace library old version with new one.

In this work, we will be using dynamic linking for reasons stated above. To learn more about DLLs, refer to [MSDN].

3.2 Interface

Every library has its API - application programming interface. API is a set of public functions, through which can user of library call functions and communicate with library. When creating C++ library, it's a good practice to create pure virtual interface as an abstract class, which is defined in header file, and then make implemented class, that is inherited from our abstract class. This way, user has access only to public interface of library, and the bodies of each function is in library file. We will also have to provide a function, that will create an instance of an implementation, and will return a pointer to it.

The abstract interface of our library is shown in Listing 3.1.

```
virtual bool initalizeParticleSystem(...) = 0;

virtual void renderParticles() = 0;
virtual void updateParticles(...) = 0;
virtual void setParticleTexture(...) = 0;

virtual UINT createPointParticleGenerator(...) = 0;
virtual UINT createSphericalParticleGenerator(...) = 0;
virtual UINT createBoxParticleGenerator(...) = 0;
virtual UINT createEllipsoidalParticleGenerator(...) = 0;

virtual bool setGeneratorColor(...) = 0;
virtual bool setGeneratorGravityVector(...) = 0;
virtual bool setGeneratorSpawnRate(...) = 0;
virtual bool setGeneratorParticleSize(...) = 0;

virtual bool setPointGeneratorPosition(...) = 0;
virtual bool setSphericalGeneratorCenter(...) = 0;
virtual bool setSphericalGeneratorRadius(...) = 0;
virtual bool setBoxGeneratorCorners(...) = 0;
virtual bool setEllipsoidalGeneratorCenter(...) = 0;
virtual bool setEllipsoidalGeneratorRadii(...) = 0;

virtual int getNumParticles() = 0;
virtual int getNumGenerators() = 0;

virtual void setProjectionMatrix(...) = 0;
virtual void setLookAt(...) = 0;

virtual bool deleteParticleGenerator(...) = 0;
virtual void clearAllParticles() = 0;
virtual void clearAllGenerators() = 0;

virtual bool releaseParticleSystem() = 0;
```

Listing 3.1: Interface of the library (... stands for actual parameters, see source code for full list)

All these functions are common for any of the implementation of particle system. No matter which one user chooses, the final effect of using these functions in the same order must cause the same effect.

3.2.1 Basic functions

The function `initializeParticleSystem` will handle all initialization of selected particle system implementation (resource allocation, shaders loading etc.). It takes one parameter, which is NULL by default - the pointer to initialization data. This parameter is unused yet, but is reserved for future development. To perform rendering of all particles, we'll call `renderParticles`. Their update is done with `updateParticles` function, which takes one parameter `fTimePassed`,

which is a time in seconds that has passed since the last rendered frame, so that particles' update speed isn't dependent on the frame rate of application, but rather on time. Function `setParticleTexture` will set the texture of a particles in a particle system. The only parameter is texture name as unsigned integer previously generated by OpenGL.

3.2.2 Generators and particles

The main idea in generating particles and achieving desired effects is to create various particle generators with different settings. Our library supports 4 types of particle generators - point, spherical, box and ellipsoidal generator. There is a function to create every generator type in the interface - `createPointParticleGenerator`, `createSphericalParticleGenerator`, `createBoxParticleGenerator` and `createEllipsoidalParticleGenerator`, respectively. Then there's a couple of setting functions. These functions will take generator ID and property that needs to be changed. The library has a built-in mechanism for detecting invalid requests, like setting radius to a point generator. If a valid request has been made, these functions return true.

User can query for number of particles using `getNumParticles` function, and for number of generators with `getNumGenerators` function.

3.2.3 Rendering-related functions

For proper translation, billboarding and rendering of particles, user must set several things. `setLookAt` function takes the same parameters as a well-known function `gluLookAt` (when using old context), or `glm::lookAt` (when using 3.1 and further context) - position of an eye, a point that eye is looking upon, and the up vector.. This is important so that we can rotate particles properly to make them face the camera. This isn't enough, user must also provide a projection matrix to library, so that the particles can be transformed into window coordinates properly. This is done using `setProjectionMatrix`. This function takes pointer to 16 floats, that represent 4x4 matrix in a column-major order (because that's how OpenGL stores matrices internally). However, this function only needs to be called when dealing with OpenGL context 3.1 and beyond, which has the fixed functionality removed. In older OpenGL contexts, these matrices were stored in OpenGL state and thus library can read them directly.

3.2.4 Deleting-related functions

To delete specific generator, user can call `deleteGenerator` function, which takes a generator ID to delete. User can also get rid of all generators with function `clearAllGenerators` and to get rid of all particles, user can call `clearAllParticles` function.

Finally there's a `releaseParticleSystem` function, which frees all used memory and de-initializes the library. This function should be called by user when he's finished working with particle system.

3.3 Particle systems

In this work, we will implement particle system using 3 different methods. First won'tuse any of OpenGL latest capabilities, but only basic rendering mechanisms from fixed functionality. This implementation will be further referred as Simple Particle System. Second will use Point sprites. The third implementation will use some of the latest GPU programming paradigms and will be done using Transform Feedback extension almost entirely on GPU. We'll also prepare some demo scenes and then make a conclusion about each implementation pros and cons. We will take into consideration number of particles and number of generators on the scene and how they influence the frame-rate. The tests will be taken on various machines with different hardware components.

3.3.1 Common properties

There are some common properties for each particle system implementation. We will go through them.

Generators

As mentioned before, there are 4 type of generators implemented in this library. All these types have however these common properties.

- **Minimal and Maximal Velocity Vector** - all generated particles by this generator will have its velocity vector taken from this range, so if VelocityMin = (-10, 2, -2) and VelocityMax = (10, 5, 2), then emmited particles will have its velocity vector in range (-10...10, 2...5, -2...2). If we set the same value to X, Y and Z component, then the velocity of all emmited particles will be the same. These values are stored in `fVelocityMin` and `fVelocityDif`, which is the difference between maximal and minimal velocity (we

remember the difference between minimum and maximum rather than maximum, so that when generating particles the difference doesn't have to be recalculated - this saves some processing time)

- **Minimal and Maximal Spawn Rate** - generator will create a bunch of particles every few seconds. Upon generator creation, next generation time is calculated. Its range will be randomly taken from minimal spawn rate to maximal spawn rate. Recall this value is NGT. After NGT seconds have passed from generator creation, new bunch of particles is created and NGT is recalculated. After NGT seconds from last recalculation, another bunch of new particles is generated and NGT is recalculated again. This goes on either forever (this is the default behavior), or only limited amount of times, if user defines it. These values are stored in `fSpawnEveryMin` and `fSpawnEveryDif`, which is the difference between maximal and minimal spawn rate

- **Size** - every generated particle will have this size. This value is stored in `fParticleSizes` variable

- **Color** - every generated particle will have this color. This value is stored in `fColor` variable

- **Minimal and Maximal Lifetime** - every generated particle will have its lifetime from this range. This values are stored in `fLifeDif`, which is the difference between maximal and minimal lifetime of particles

- **Minimal and Maximal Count of Particles** - every time generator generates a particles (the NGT seconds have passed since last recalculation), the number of particles generated will be in this range. These values are stored in in `countMin` and `countDif`, which is the difference between maximal and minimal count of generated particles

- **Maximal Number of Spawns** - this means maximal number of spawning a bunch of particles, or maximal number of NGT recalculations, before the generator expires. The default value of this parameter is -1, which means generate particles indefinitely. This value is stored in in `maxSpawns` variable. The values of spawns that has already occured is stored in `numSpawns` variable

- **Gravity Vector** - gravity vector alters velocities of each particle generated by this generator. The default value is zero gravity vector, so particles are not influenced by this. This is useful for creating effects like fountains. This value is stored in `fGravityVector` variable

Other common properties are `fNextTime`, which is the time of next generation of new particles. `fPassedTime` tracks the time, starting from 0.0 and if it passed `NGT`, that's stored in `fNextTime`, new bunch of particles is generated. Integer `iNewCount` stores the number of particles that will be generated next time.

Because all the above properties are common, we will define a base class, from which all other generator classes will be inherited. You can see how generator base class looks like in Listing 3.2.

```
class PSE_ParticleGenerator
{
public:
  void initGenerator(float* a_fVelocityRangeMin, float*
      a_fVelocityRangeMax, float a_fSpawnEveryMin, float
      a_fSpawnEveryMax, int a_countMin, int a_countMax, float
      a_fLifeMin, float a_fLifeMax, float a_fParticleSizes, int
      a_maxSpawns = -1);

  void regenerateNextTime();
  void setGeneratorColor(BYTE a_r, BYTE a_g, BYTE a_b);
  void setGravityVector(float* a_fGravity);
  void setSpawnRate(float fNewSpawnEveryMin, float fNewSpawnEveryMax,
      int iNewMinCount, int iNewMaxCount);
  void setParticleSize(float a_fNewParticleSize);
  bool isDead();
};
```

Listing 3.2: Base class of generator (only interface part)

Most functions are self-explanatory, except `regenerateNextTime`, which calculates next spawning time and number of spawned particles in next generation, and `isDead`, which returns true, if particle generator performed maximum number of particle generations.

Next we'll examine different generator types.

- **Point generator**

  This generator is the simplest and has one additional property. It's generator's position. All generated particles are created at the same position.

- **Spherical generator**

  This generator has two additional properties - center and sphere radius. Now the generated particles' initial positions are taken randomly from within the sphere, not only from its surface. The sphere generator with a radius 0.0 is a point generator.

- **Box generator**

  This generator has two parameters - arbitrary opposite corners of a desired 3D bounding box. New particles' positions are randomly taken from the inside of a specified bounding box.

- **Ellipsoidal generator**

  This generator has 4 additional properties - ellipsoid center and its radii in X, Y and Z direction, respectively. Generated particles' position are randomly taken from the inside of the ellipse.

Particles

Every generated particle has following properties: position, velocity vector, gravity vector, color, lifetime left and size. Additionally, particles will have extra properties depending on particle system implementation, as will be shown further in this work. The base class of particle is shown in Listing 3.3.

```
class PSE_Particle
{
public:
  float fPosition[3];
  float fVelocity[3];
  float fGravityVector[3];
  float fColor[3];
  float fLifeTimeSpan;
  float fSize;
};
```

Listing 3.3: Base class of particle

Velocity means by how much will particle's position be updated when 1 second passes. Lifetime is also in seconds and holds time before the particle expires. Gravity vector is the amount added to velocity vector of particle after 1 second.

Memory management

Each of our particle system implementations will have dynamic array of generators. We'll use STL vectors for this. More specific, we'll have a vector of pointers to generators. Everytime, when user wants to create a generator, the system creates it somewhere in memory and adds pointer to the vector of generators. System also must return generator ID and give it to user, so that he can

work with it later. This ID will actually be index in the pointer array. The newly generated ID can be one of the followings:

- either all the pointers in the array are active and their generators are working and thus the new generator ID will be index of latest added pointer
- or some of the pointers are "dead" by now, i.e. have expired or been deleted by user, and system retrieves the first index where NULL pointer is set

System internally remembers first unused index (further referred as `FUI`) and with each newly created or deleted generator this index is recalculated. When a generator is deleted, the `FUI` is a minimum from the deleted generator ID and value stored in `FUI`. The pointer at that place is then deleted and set to NULL. When a generator is created, it's placed on `FUI` index (in case `FUI` is equal to the size of dynamic array of pointers, the array is resized by one). System then scans all further indices and sets the `FUI` to the first deleted generator (where pointer is NULL), or to the end of dynamic array (its size), whichever comes first. That's why `FUI` recalculation runs in $O(N)$ time, where N is size of dynamic array, which is practically number of generators in the particle system, and the addition of new generator runs in $O(1)$ time with assumption that memory allocation using `new` C++ keyword runs in constant time. There is a room for improvement of this simple memory management, however this part isn't crucial for the particle system and doesn't affect the framerate and overall performance of system significantly.

Because this is common for all implementations, there is a derived class from `PSE_ParticleSystemEngineCommon`, which implements this memory management and is a base class for subsequent implementations. This class also implements generators altering functions and setting of projection matrix and view properties. However, deleting of generators isn't implemented in this class, but rather left for derived class to implement it, because deletion of generators on GPU and server side differs from the deletion on CPU and client side.

3.3.2 Simple particle system

This implementation will do all particle calculations and updating on CPU. Rendering of particles is done using immediate rendering mode.

Rendering

We will be rendering and updating particles sequentially one by one. To achieve this, we will create dynamic array that will store all the active particles with their properties. We'll use STL vectors for this as well. The rendering part is done using quads rendering mode in OpenGL. With each particle, we must send 4 vertices to the GPU. Texture coordinates are static, for each point of quad there is a texture coordinate, concretely (0.0, 1.0) for upper-left corner, (1.0, 1.0) for upper-right corner, (1.0, 0.0) for lower-right corner and (0.0, 0.0) for lower-left corner. The vertex positions however must be calculated, depending on position and view vector of camera. Everytime a user sets a view vector of camera (calls `setViewVector` function), two vertices, called `vQuad1` and `vQuad2` are calculated using cross product. These vectors are perpendicular to each other and they define 2D plane on which the particles should be rendered. For each particle, 4 vertex positions must be calculated using these two vectors and particle size, as shown in figure 3.1.

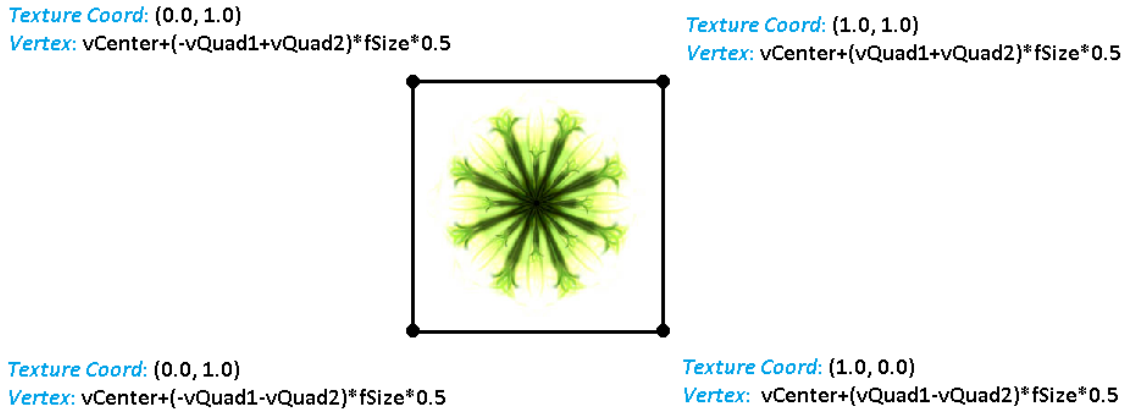

Figure 3.1: Calculation of particle vertices

For more detailed explanation of billboarding, refer to [BillboardingTut].

Before rendering takes place, we enable blending and disable writing to the depth buffer. Blending is turned on to make particles transparent depending on their lifetime - if a particle has more than 1 second of lifetime left, the alpha of it's color is set to 1.0. However, if particle has lifetime below 1 second, alpha of it's color is set to its lifetime, creating a nice fadeout effect.

Depth buffer writing is disabled, because not doing so would make particles' rendered quads clearly visible and some particles would not even get drawn, because they may appear behind already rendered particle, which has already updated depth buffer values even though it is transparent and particles behind it should be seen and taken into consideration when calculating final color in the framebuffer.

Updating

Updating takes whole place on CPU. First we'll go through all the generators and call their `generateParticles` function. Each generator type implements this in its own way. Additionally, we do a check whether generator is "dead", i.e. it's already reached it's maximal number of generations of particles. If so, the generator is deleted.

After generating phase, we'll go through all particles sequentially, one by one, and update their properties. Particle's velocity is scaled depending on passed time and added to its position. Lifetime of particle is decreased by the time passed since the last frame. If the lifetime drops below 0.0, then the particle has exhausted and is deleted. Deletion of particle isn't done using STL vector `erase` function, because it runs in linear time, but we do this in $O(1)$ time. The trick is to copy the particle that's on the last index of the array to the index of particle being deleted and then decreasing size by 1 using `pop_back` function. Also, the gravity vector (scaled by time elapsed since the last frame) is added to the velocity vector of particle. Note that the gravity vector is also stored as a particle property. Everytime generator creates a particle, it copies its actual gravity vector to particle. This is because user can change the gravity vector of generator anytime he wants, so we must remember this value for every particle.

Updating particles on CPU side can be improved using multi-threading, where each thread updates one particle at time. This would however require some synchronization objects built into system to achieve mutual exclusion.

To retrieve the number of particles in the system, we simply check the current size of particles' dynamic array. To get the number of generators in system, we must traverse through all pointers and count in only non-nullified ones. Clearing generators and particles here is also trivial.

3.3.3 Point sprites particle system

This implementation improves the rendering of simple particle system. Point sprites is a feature adopted to core OpenGL in version 1.5. It's listed as `GL_ARB_POINT_SPRITE` and can be found

at [?]. Although OpenGL has always supported texture mapped points, prior to version 1.5 this meant a single texture coordinate applied to an entire point. Large textured points were simply large versions of a single filtered texel. But with point sprites, it is possible to place a 2D textured image anywhere onscreen by drawing a single 3D point. Point sprites allow to render a perfectly aligned textured 2D square by sending a single 3D vertex. This reduces the bandwidth of sending vertices by 4, since we don't have to pass 4 vertices now like in previous case. The size of rendered square is dependent on several settings of point sprites. Also there's no need to send down texture coordinates anymore, because they're calculated automatically. This saves additional processing time. To perform render, we'll use immediate render mode, while sending only vertex positions to GPU.

Rendering

The main setting of point sprites implemenation is point size attenuation depending on point from camera distance. We define 3 types of attenuations - constant, linear and exponential. We must also tell OpenGL, that we want to use point sprites. This is done by calling `glEnable(GL_POINT_SPRITE_ARB)`. We also must tell it to replace the texture coordinates with point sprites texture coordinates by changing the texture environment. It's done with calling `glTexEnvf(GL_POINT_SPRITE_ARB, GL_COORD_REPLACE_ARB, GL_TRUE)`. Because each particle has its own size, we must set it as well with each particle rendered. The problem is, that size setting function `glPointSize` cannot be called between `glBegin` and `glEnd` functions. So instead of changing the size from particle to particle we'll rather remember last rendered particle size. If it's same, we don't have to do anything, otherwise we'll stop drawing, change the size and then continue it.

As in simple particle system, we must enable blending and disable writing to the depth buffer for the same reasons.

Updating

Updating part is exactly the same as in simple particle system.

3.3.4 Transform feedback particle system

This implementation will be completely different from the previous two and will use the latest GPU programming paradigms. It will rely on shaders and the transform feedback OpenGL extension, which comes with OpenGL 3.0 and further. This implementation won't use any of the fixed pipeline from the previous two implementations (they're marked as deprecated in OpenGL 3.0 and are removed from OpenGL 3.1). The previous two implementations relied heavily on the CPU - most of the work has been done there, while GPU was waiting for CPU to finish its job before it could render. But this implementation will leave most of the work on the GPU, and CPU will only control GPU. Because modern GPUs have several shader units, the processing of particles can run parallel, as on CPU we were doing this sequentially.

Transform feedback

Transform Feedback is the process of altering the rendering pipeline so that primitives processed by a Vertex Shader and optionally a Geometry Shader are written to buffer objects. This allows to preserve the post-transform rendering state of an object and resubmit this data multiple times. This feature exposes GPU parallelism possibilities easily and allows us to put any kind of work to GPU and then read the results of the work from the VBO that transform feedback has filled. This is ideal for creating particle system. It's listed as `GL_ARB_transform_feedback3` and can be found at [TransFB].

The idea is to create two shader programs - one for particle generation and updating (referred further as generator program), and one for rendering. The particle generation program will consist of vertex and geometry shader and no fragment shader, because we aren't rendering yet. Vertex shader will process the vertices, which will actually be particles and vertex attributes will be associated particle attributes. These vertices will be further sent to geometry shader, which will do the updating and will emit as many vertices, as many alive particles there are. The second shader program will process only rendering of particles.

In OpenGL, transform feedback is treated as an object, so first we must generate one and remember its name (unsigned int assigned by OpenGL). This is done using `glGenTransformFeedbacks`, which takes as parameter number of transform feedback objects we want to generate and where to store them. For whole particle system, we need only 1 transform feedback object. In order for transform feedback to work, we must tell it where to store the recorded data. This is done by function `glBindBufferBase`. We will pass there name of

vertex buffer object, where the data from transform feedback will be recorded. This buffer must be large enough to be able to store all the data generated by geometry shader. These data will be reused for rendering next frame. In our case, every generator will have two VBOs - one used for reading of output particles and one for writing of particles. The maximum number of particles on the scene at once for a given generator is calculated as follows:

```
int iMaximumParticles = countMin+int(((fLifeMin+fLifeDif)/
   fSpawnEveryMin)*float(countMin+countDif));
```

This assumes the worst case scenario, that random number function will return maximum count of particles with maximal lifetime as often as possible. In average case, this array will never be used at its full capacity. But it's large enough to hold all particles. We must define one additional property for particles on GPU - whether its a normal particle, or generator. What we want is that the generator program generates particles exactly once. Because generator is coming to generator program geometry shader as an ordinary particle, we need to differentiate it from other particles. That's why we'll use inherited class `PSE_ParticleGPU`, which will also have type in addition to other particle properties, as shown in listing 3.4.

```
class PSE_ParticleGPU : public PSE_Particle
{
public:
  int iType;
};
```

Listing 3.4: GPU particle class

Type 0 will represent a generator, and type 1 will represent particle. With this we'll make sure, that all the particles that come into our generator program will be processed with their values, and exactly once the shader will check if it's time to generate particles and, in case, generate them, because there will always be exactly one incoming particle of type 0.

Because generators in this implementation differ from the generators that were running only on CPU, we will create inherited class `PSE_ParticleGeneratorGPU`, which will wrap all additional properties, as shown in listing 3.5.

```
class PSE_ParticleGeneratorGPU : public PSE_ParticleGenerator
{
public:
  void initGPUResources();

  void useProgram();
  virtual void setSpecificGeneratorParameters() = 0;
  void generateParticles(float fTimeElapsed);
  void renderParticles();
  void setShaderPrograms(CShaderProgram* a_spGeomGen, CShaderProgram*
      a_spRender);
  void setQueryID(UINT a_uiQuery);
  int getNumParticles();

  void requestDelete();
  bool canBeDeleted();

  void deInitGenerator();
  void clearAllParticles();
};
```

Listing 3.5: GPU particle generator class (only interface part)

All the initalization specific for GPU generators is done in function `initGPUResources`. We allocate VBO large enough to hold all particles and create a VAO object that describes data layout of particles in VBO. VBO is initalized using function `glBufferData` with proper size, but NULL pointer specified. This call just allocates memory on GPU. The drawing hint is `GL_DYNAMIC_DRAW` which hints OpenGL driver, that the content of this buffer will be changed often, so that it can optimize the performance. Additionally, the first particle, that is actually a generator, because it's type is set to 0, is uploaded to VBO, using function `glSubBufferData`.

Function `setShaderProgram` sets generator program name. This is important, so that generator can set program's uniform variables. The only abstract function `setSpecificGeneratorParameters` sets all the properties that are specific for each generator type, like radius of a spherical generator. For retrieval of particle count, we must use a query object, as will be explained later. But it's enough to have one global query object, created by particle system implementation and then passing it to each generator. This is done using function `setQueryID`. Three functions deal with particle and generator deletion - `requestDelete`, `canBeDeleted` and `clearAllParticles`. Their purpose is explained later. Function `deInitGenerator` serves as function that frees all used memory on GPU (VBO and VAO).

Generator program

As mentioned before, generator program will consist only of vertex and geometry shader. Vertex shader will simply pass incoming data to geometry shader. It is shown in listing 3.6.

```
#version 330

layout (location = 0) in int iType;
layout (location = 1) in vec3 vPosition;
layout (location = 2) in vec3 vVelocity;
layout (location = 3) in vec3 vColor;
layout (location = 4) in float fLifeTime;
layout (location = 5) in float fSize;

out int iTypePass;
out vec3 vPositionPass;
out vec3 vVelocityPass;
out vec3 vColorPass;
out float fLifeTimePass;
out float fSizePass;

void main()
{
  iTypePass = iType;
  vPositionPass = vPosition;
  vVelocityPass = vVelocity;
  vColorPass = vColor;
  fLifeTimePass = fLifeTime;
  fSizePass = fSize;
}
```

Listing 3.6: Generator program vertex shader

The geometry shader will be responsible for particles updating and generation. However, we must tell it how many seconds have passed since the last frame in order to function properly. This is done using uniform variables. Additionally, all generator properties are stored as uniform variables. Point generator geometry shader is shown in listing 3.9. Other types of generators have similar geometry shaders, but with different particle generating algorithms. However, all common properties of generators and updating of particles is defined in other shader files, because they are the same and they are added to shaders using `#include` keyword. This is not official GLSL keyword, however we implemented this in the `loadShader` function to actually copy content of another file and paste it into location of include (similar as in C++). In the listing 3.7 you can see common gemerator shader program properties, that are included into each generator geometry shader. There is also our own random function, because GLSL doesn't have one. From the client side, we only need to set random number generator seed, so that it generates different numbers

each frame. It returns floating point value from 0.0 to 1.0.

```
layout(points) in;
layout(points) out;
layout(max_vertices = 40) out;

in vec3 vPositionPass[];
in vec3 vVelocityPass[];
in vec3 vGravityPass[];
in vec3 vColorPass[];
in float fLifeTimePass[];
in float fSizePass[];
in int iTypePass[];

out vec3 vPositionOut;
out vec3 vVelocityOut;
out vec3 vGravityOut;
out vec3 vColorOut;
out float fLifeTimeOut;
out float fSizeOut;
out int iTypeOut;

uniform float fDeltaTime;
uniform vec3 vGenVelocityMin, vGenVelocityDif;
uniform vec3 vGenColor;
uniform vec3 vGenGravityVector;
uniform float fLifeMin, fLifeDif;
uniform float fTimeLeft;
uniform int iNumToGenerate;
uniform float fGenSize;
uniform int iGenerate; // It's used as bool

vec3 vLocalSeed;
uniform vec3 vSeed;

float rand()
{
    uint n = floatBitsToUint(vLocalSeed.y * 214013.0 + vLocalSeed.x *
        2531011.0 + vLocalSeed.z * 141251.0);
    n = n * (n * n * 15731u + 789221u);
    n = (n >> 9u) | 0x3F800000u;
    float fRes =  2.0 - uintBitsToFloat(n);
    vLocalSeed = vec3(vLocalSeed.x + 147158.0 * fRes, vLocalSeed.y*
        fRes  + 415161.0 * fRes, vLocalSeed.z + 324154.0*fRes);
    return fRes;
}
```

Listing 3.7: Common generator properties in geometry shader

In the listing 3.8 there's a code that's common for all generators and performs updating of each particle.

```
vLocalSeed = vSeed;

vPositionOut = vPositionPass[0];
vVelocityOut = vVelocityPass[0];
if(iTypePass[0] != 0)vPositionOut += vVelocityOut*fDeltaTime;

if(iTypePass[0] != 0)vVelocityOut += vGravityPass[0]*fDeltaTime;
vGravityOut = vGravityPass[0];
vColorOut = vColorPass[0];
fLifeTimeOut = fLifeTimePass[0]-fDeltaTime;
fSizeOut = fSizePass[0];
iTypeOut = iTypePass[0];
```

Listing 3.8: Common generator update code

The geometry shader will work as follows. First, it checks whether incoming particle is a generator. If so, it will emit it back into particle stream (store it in VBO) using function `EmitVertex` and `EndPrimitive`, because each primitive is now only a single point, which represents a particle. Then there's a check if it isn't time to generate particles. If it is, it will emit as many particles as it has this time. However, if the generator has expired (reached its limit of generating times), it shouldn't generate more particles. That's why there's `iGenerate` uniform variable, which is used as bool, to tell shader not to emit particle anymore.

If the incoming particle wasn't generator, but only normal particle, geometry shader will do all the necessary particle updates using particle data that came from vertex shader. Shader will re-emit particle only if it's lifetime isn't over yet. If it is, shader simply won't emit the vertex and in the next frame, the particle won't be there. This way, we have put the whole updating process on GPU. CPU only tells all important data, like time passed from the last frame and generator properties.

```
#include "generator_common.geom"

uniform vec3 vGenPosition;

void main()
{
  #include "generator_update.geom"

  if(iTypeOut == 0)
  {
    EmitVertex();
    EndPrimitive();

    if(fTimeLeft < 0.0 && iGenerate == 1)
    {
      for(int i = 0; i < iNumToGenerate; i++)
      {
        vPositionOut = vGenPosition;
        vVelocityOut = vGenVelocityMin+vec3(vGenVelocityDif.x*rand(),
           vGenVelocityDif.y*rand(), vGenVelocityDif.z*rand());
        vGravityOut = vGenGravityVector;
        vColorOut = vGenColor;
        fLifeTimeOut = fLifeMin+fLifeDif*rand();
        fSizeOut = fGenSize;
        iTypeOut = 1;
        EmitVertex();
        EndPrimitive();
      }
    }
  }
  else if(fLifeTimeOut > 0.0)
  {
      EmitVertex();
      EndPrimitive();
  }
}
```

Listing 3.9: Point generator geometry shader

To make the transform feedback work, we must tell OpenGL which data we want to record from transformation process and where to store them. First task is done using function `glTransformFeedbackVaryings`, where we specify names of output variables from geometry shader, given its present, or vertex shader. This function must be called before the program is linked, so that OpenGL can check whether all required variables are present ion shader program. The second task is done using function `glBindBufferBase`, which takes a name of VBO and adds it to currently bound transform feedback object on specified index. Index is used if we wanted to record data to more than just one VBO. But in this case, we need data to be recorded only into

one VBO, so this index is alwasy 0. This buffer must be assigned for each generator separately, as they all have their own for storing particles.

If the particle updating is being done first time for a given generator, the input data stored in VBO contain only one particle with type 1. We proceed with "rendering" one point - particle generator, to start particle updating process. The next frame, this buffer will contain data from transform feedback. The number of output particles can be queried using OpenGL query object by calling `glGetQueryResultiv`. This way we'll know the number of particles that have been recorded to the VBO. When updating particles, we don't want rasterization to be enabled, so we disable it using `glEnable(GL_FRAGMENT_RASTERIZATION_DISCARD)`.

If a user wants to delete a generator, it cannot be done right away in this implementation, because generator object may still have some particles in the scene, that are alive. So instead of immediately deleting generator, we rather set a flag that generator should be deleted, and as soon as all the particles from this generator fade away, we can safely delete it. The request is done using `requestDelete` function. When a flag to delete generator is set, `iGenerate` variable is set as well to prevent shader program from emitting additional particles. When the generator has a request for deletion and number of its particles is 0, then it can be really deleted and used memory can be freed. All these checks are done in function `canBeDeleted`.

To retrieve the number of particles for specific generator, we just have to look for number of recorded particles in VBO, and subtract 1 from it because of the generator present in VBO as particle with type 0. Clearing particles is done differently now, what we do is that we set the number of particles that were recorded by transform feedback to 1, so that we leave only generator there.

Rendering

Rendering is done in a similar manner as on CPU. However, we won't be calculating anything on client side, like billboarding in a simple particle system. This time, we have all particles and their associated data stored in generator's VBO. We also know their count. So the only thing left to do is to set rendering shader program, and begin rendering particles as points by calling `glDrawArrays(GL_POINTS, 0, iNumParticles)`. The rendering program will be same for all generator types, because it only deals with already updated particles rendering. Before rendering takes place, we must set the modelview and projection matrix to this program, as well as the vertices of a billboarding plane. We also must re-enable rasterization using

`glDisable(GL_FRAGMENT_RASTERIZATION_DISCARD)`.

The rendering program now has all 3 type of shaders - vertex, geometry and fragment. Vertex shader is responsible only for passing data of particles further. It differs from the vertex shader of generator program in only 2 things - there is no velocity and gravity vector of particles, because they are important only for updating, not rendering. Vertex shader is shown in listing 3.10.

```
#version 330

layout (location = 0) in int iType;
layout (location = 1) in vec3 vPosition;
layout (location = 3) in vec3 vColor;
layout (location = 4) in float fLifeTime;
layout (location = 5) in float fSize;

out float fSizePass;
out float fLifeTimePass;
out int iTypePass;
out vec3 vColorPass;

void main()
{
    gl_Position = vec4(vPosition, 1.0);
    fSizePass = fSize;
    fLifeTimePass = fLifeTime;
    iTypePass = iType;
    vColorPass = vColor;
}
```

Listing 3.10: Vertex shader of rendering program

With these data, geometry shader can continue with creating billboarded quads. However, quads rendering mode is no longer supported, so we'll use triangle strip instead, but change the order of emitted vertices. To render a quad using triangle strip, we'll use the vertex ordering as shown in figure 3.2.

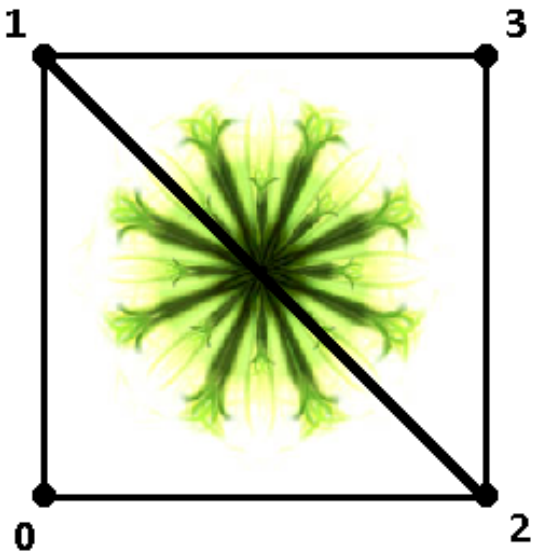

Figure 3.2: Order of vertices in TRIANGLE_STRIP render mode

The input type of primitive to geometry shader is point, and the output type is triangle strip. The geometry shader is shown in listing 3.11.

```
#version 330

uniform struct Matrices
{
  mat4 mProj, mModel, mView;
} matrices;

uniform vec3 vQuad1, vQuad2;

layout(points) in;
layout(triangle_strip) out;
layout(max_vertices = 4) out;

in float fSizePass[];
in float fLifeTimePass[];
in int iTypePass[];
in vec3 vColorPass[];

smooth out vec2 vTexCoord;
flat out vec4 vColorPart;

void main()
{
  if(iTypePass[0] != 0)
  {
    vec3 vPosOld = gl_in[0].gl_Position.xyz;
    float fSize = fSizePass[0];
    mat4 mVP = matrices.mProj*matrices.mView;

    vColorPart = vec4(vColorPass[0], fLifeTimePass[0]);

    vec3 vPos = vPosOld+(-vQuad1-vQuad2)*fSize;
    vTexCoord = vec2(0.0, 0.0);
    gl_Position = mVP*vec4(vPos, 1.0);
    EmitVertex();
    vPos = vPosOld+(-vQuad1+vQuad2)*fSize;
    vTexCoord = vec2(0.0, 1.0);
    gl_Position = mVP*vec4(vPos, 1.0);
    EmitVertex();
    vPos = vPosOld+(vQuad1-vQuad2)*fSize;
    vTexCoord = vec2(1.0, 0.0);
    gl_Position = mVP*vec4(vPos, 1.0);
    EmitVertex();
    vPos = vPosOld+(vQuad1+vQuad2)*fSize;
    vTexCoord = vec2(1.0, 1.0);
    gl_Position = mVP*vec4(vPos, 1.0);
    EmitVertex();
    EndPrimitive();
  }
}
```

Listing 3.11: Geometry shader of rendering program

The last part is only fragment shader, which only applies colored texture into the final image. It's shown in listing 3.12.

```
#version 330

uniform sampler2D gSampler;

smooth in vec2 vTexCoord;
flat in vec4 vColorPart;

out vec4 FragColor;

void main()
{
  vec4 vTexColor = texture2D(gSampler, vTexCoord);
  FragColor = vec4(vTexColor.xyz, 1.0)*vColorPart;
}
```

Listing 3.12: Fragment shader of rendering program

3.4 Effects library

Along with the particle system library, we created a library with some pre-programmed effects. This library simplifies creation of effects in the user application even more, because user just needs to call a single function to create a fire effect for example. The interface of effects library is shown in listing 3.13.

```
namespace BlazeEffects
{
  void setParticleSystemEngine(...);

  UINT createFire(...);
  UINT createFireRing(...);
  UINT createFireGrid(...);

  bool deleteEffect(...);

  bool deleteAllEffects();
};
```

Listing 3.13: Interface of effects library (... stands for actual parameters, see source code)

Using this library is easy - only thing we need to do is set already existing Blaze particle system engine object with function `setParticleSystemEngine`. There are currently 3 available effects now - fire, fire ring, and fire grid, created with functions `createFire`, `createFireRing` and

`createFireGrid`, respectively. All these methods return an identifier of effect. Using this identifier, user can later delete the effect by calling `deleteEffect` with previously generated effect ID. To delete all effects at once, user can call `deleteAllEffects` function.

Chapter 4

Evaluation

In this chapter, we'll cover and explain the usage of the library. We'll create 3 different demo scenes, each with miscellaneous generators and settings. All 3 demos will be able to be run with all 3 implementations of the particle system. We will also compare the performance of each implementation at specific demos. These demos will be tested on 3 different machines:

Low-end machine:

- Processor: Intel Core 2 Duo
- Memory: 4GB Ram DDR2
- Graphics Card: nVidia GeForce 9300m
- Operating System: Windows Vista

Mid-end machine:

- Processor: Intel Core 2 Quad
- Memory: 4GB Ram DDR2
- Graphics Card: nVidia GeForce GT440
- Operating System: Windows 7

High-end machine:

- Processor: Intel Core i5 Quad

- Memory: 8GB Ram DDR2
- Graphics Card: AMD Radeon HD5850
- Operating System: Windows 7

The demos will run in 1440x900 resolution.

4.1 Fire Demo Scene

This scene contains one fireplace, 3 fire rings, and a grid of fire. The average number of particles in this scene is around 30000.

Initialization

First, we need to initialize our particle system library in the `initScene` function of demo. After that, we'll call functions from the Blaze effects library to create desired effects. When this is done, we'll set fire texture to the particle system.

```
void prepareFireDemoScene()
{
  BlazeEffects::setParticleSystemEngine(psEngine);

  BlazeEffects::createFire((float*)glm::value_ptr(glm::vec3(148, 0.0f,
      -148.0f)), (float*)glm::value_ptr(glm::vec3(4.0f, 0.0f, 4.0f)),
     200, 64, 0);
  BlazeEffects::createFireRing((float*)glm::value_ptr(glm::vec3(-50.0f
     , 0.0f, 50.0f)), 20.0f, 20, 51, 189, 255);
  BlazeEffects::createFireRing((float*)glm::value_ptr(glm::vec3(50.0f,
      0.0f, -150.0f)), 30.0f, 30, 255, 128, 0);
  BlazeEffects::createFireRing((float*)glm::value_ptr(glm::vec3(-150.0
     f, 0.0f, -150.0f)), 10.0f, 15, 255, 0, 0);
  BlazeEffects::createFireGrid(glm::vec3(0.0f, 28.0f, 120.0f), 50.0f,
     4, 255, 128, 0);

  psEngine->setParticleTexture(tFireTexture.getTextureID());
}
```

Listing 4.1: Initialization of fire demo scene

Rendering

The actual rendering is in the `renderScene` function, and it consists of these function calls:

```
void renderScene(COpenGLControl* oglControl)
{
  //...

  psEngine->setProjectionMatrix(glm::value_ptr(*oglControl->
     getProjectionMatrix()));
  glm::mat4 mLook = cCamera.look();
  psEngine->setLookAt(cCamera.vEye.x, cCamera.vEye.y, cCamera.vEye.z,
     cCamera.vView.x, cCamera.vView.y, cCamera.vView.z, cCamera.vUp.x,
      cCamera.vUp.y, cCamera.vUp.z);

  float fTimePassed = appMain.sof(1.0f);
  psEngine->updateParticles(fTimePassed);
  psEngine->renderParticles();

  //...
}
```

Listing 4.2: Rendering with Blaze particle system

The setting of matrices is needed only if we're using shaders to perform rendering. In our case it's when using transform feedback particle system engine and OpenGL Context 3.3. Rendering is common for all demo scenes, so it's listed only in listing 4.2.

Results

In the table 4.1, you can clearly see, that transform feedback implementation runs best - it's number of frames generated per second (FPS) is highest.

Implementation / Test Machine	Low-end	Mid-end	High-end
Simple	60 FPS	76 FPS	121 FPS
Point Sprites	56 FPS	42 FPS	68 FPS
Transform Feedback	48 FPS	51 FPS	81 FPS

Table 4.1: Fire demo scene evaluation

The screenshot from fire demo scene is shown in figure 4.1.

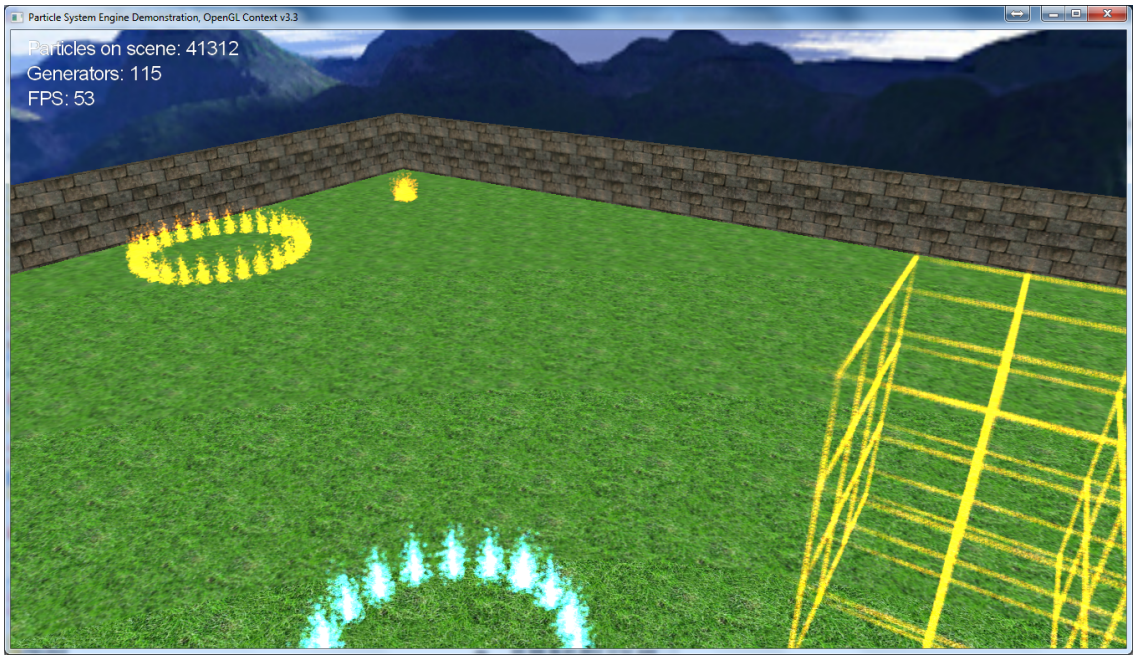

Figure 4.1: Fire demo screenshot

4.2 Space Demo Scene

This scene is in space and user can shoot "missiles" with left mouse button, which explode after a little time. The average number of particles in this scene is around 7000.

Initialization

Initialization consists only of creating a path of blue particles.

Results

In the table 4.2, you can see the average number of FPS with each implementation.

Implementation / Test Machine	Low-end	Mid-end	High-end
Simple	60 FPS	206 FPS	217 FPS
Point Sprites	52 FPS	254 FPS	268 FPS
Transform Feedback	47 FPS	70 FPS	117 FPS

Table 4.2: Space demo scene evaluation

The screenshot from fire demo scene is shown in figure 4.2.

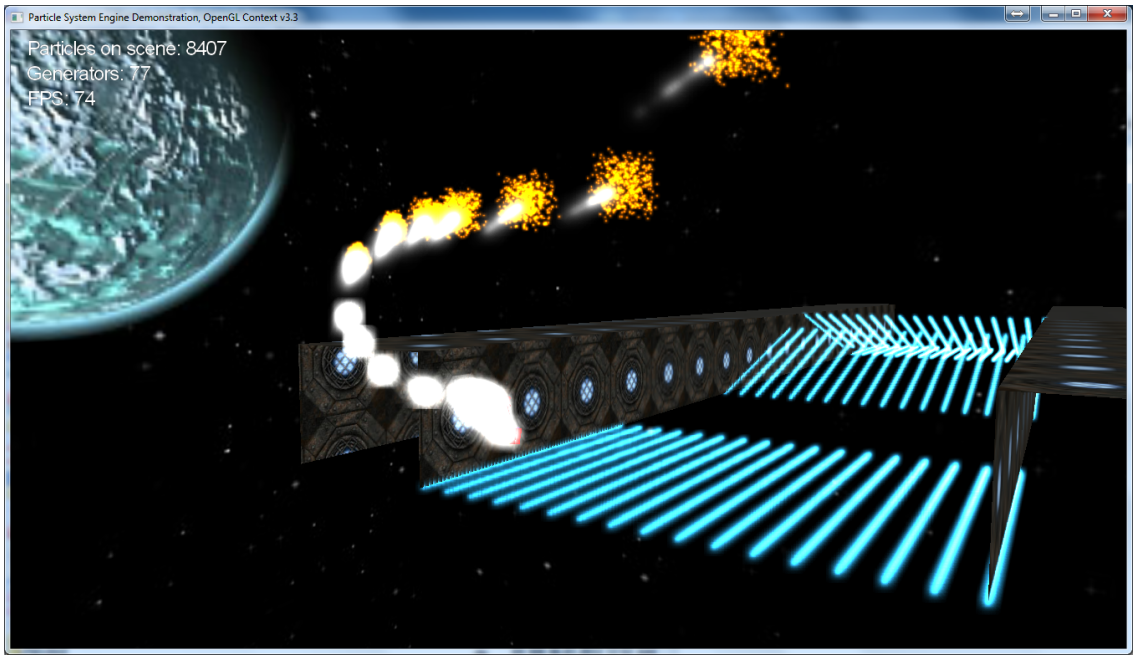

Figure 4.2: Space demo screenshot

4.3 Fountain Demo Scene

This scene contains a pool with fountain. Average number of particles here is 60000, although there is a small number of generators.

Initialization

Initialization consists of creating a spherical generator in the middle with large particle generation rate and several smallers around it.

Results

In the table 4.3, you can see the average number of FPS with each implementation.

Implementation / Test Machine	Low-end	Mid-end	High-end
Simple	58 FPS	85 FPS	114 FPS
Point Sprites	31 FPS	46 FPS	80 FPS
Transform Feedback	41 FPS	88 FPS	127 FPS

The screenshot from fire demo scene is shown in figure 4.3.

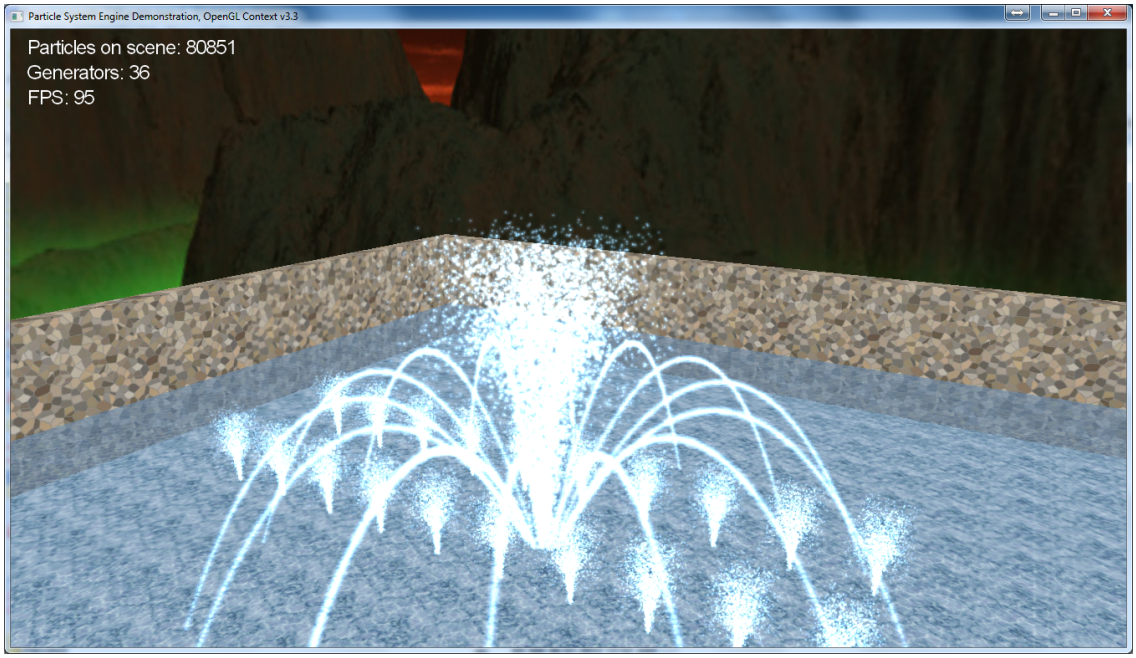

Figure 4.3: Fountain demo screenshot

Conclusion

In this work we implemented particle system both on CPU and GPU using OpenGL. We could clearly see, that modern GPUs excel at parallelization tasks, because of many independent shader units they have. Particle updating and rendering belong to such tasks. However, with too many generators on scene, our GPU implementation suffered from constant data source rebindings, causing a slowdown, whereas CPU implementations didn't have such problems, because they're not making any changes when dealing with data from multiple generators, because they all shared common buffers for particle storage. But for normal usage, our GPU implementation is definetely a lot faster than CPU one.

Future work

The development of this library will continue with adding new features into it. All generators should have their own transformation matrix, so that they don't have to be aligned only along X, Y and Z axis. Also there are features in OpenGL 4.0 and beyond, that extend transform feedback functionality, like pausing and resuming it, and could push the performance of transform feedback implementation even further. CPU implementation can be upgraded to multi-threaded, which should improve performance approximately by number of cores on processor. Particles will be influenced by more forces, so that more complex physical simulations will be possible.

Bibliography

[REE83] W. T. Reeves. "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects", *Computer Graphics 17:3 pp. 359-376*, 1983 (SIGGRAPH 83)

[REE85] W. T. Reeves. "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems", *Computer Graphics, vol. 19, no. 3, pp 313-322*, 1985

[REY87] C. W. Reynolds. "Flocks, Herds, and Schools: A Distributed Behavioral Model", *Computer Graphics, vol. 21, no. 4, pp 25-34*, 1987

[TFtut] `http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html`
Transform Feedback Particle System OpenGL Tutorial

[BLAZE] `www.mbsoftworks.sk/blaze/index.php`
Blaze Particle System Library Webpage

[FLINT] `http://flintparticles.org`
Blaze Particle System Library Webpage

[SPARK] `http://spark.developpez.com/index.php`
SPARK Particle System Library Webpage

[VertArray] `http://mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=6`
Vertex Array Objects Tutorial

[BillboardingTut] `http://www.lighthouse3d.com/opengl/billboarding/billboardingtut.pdf`
Billboarding Tutorial

[GLregistry] `http://www.opengl.org/registry`
OpenGL Registry

[PointSprites] http://www.opengl.org/registry/specs/ARB/point_sprite.txt

Point sprites overview

[TransFB] http://www.opengl.org/registry/specs/ARB/transform_feedback3.txt Transform feedback overview

[GLSLref] http://www.opengl.org/documentation/glsl

OpenGL Shading Language Reference Pages

[GLM] http://glm.g-truc.net/

OpenGL Mathematics Library

[Khronos] www.khronos.org

Khronos Group

[MSDN] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682589(v=vs.85).aspx

Dynamic Link Libraries Overview