General-Purpose Computing on GPUs

Dirk Vanden Boer

Promoter: Frank Van Reeth
Supervisor: Tom Van Laerhoven

Final dissertation submitted for obtaining the degree of master in computer science, graduate variant Multimedia

School of Information Technology
Transnationale Universiteit Limburg
Diepenbeek, June 6, 2005

Abstract

GPGPU stands for General-Purpose computation on GPUs. With the increasing programmability of commodity graphics processing units (GPUs), these chips are capable of performing more than the specific graphics computations for which they were designed. They are now capable coprocessors, and their high speed makes them useful for a variety of applications [1].

We give a brief history of the GPU to see how it has developed over the last couple of years, discuss the benefits and drawbacks of using the GPU and show why it is so useful for solving general problems. We also take a look at what the future GPUs will probably look like. To get some insight of how programming on the GPU works we explain the stream programming model used to program on GPUs and discuss the operations that are available on the GPU and how to deal with operations that are not available. Also, some mechanisms are presented that are often used to convert algorithms from the Central Processing Unit (CPU) to the GPU.

To give an idea of the applicability of GPGPU we give an overview of the most important research areas in which the GPU can be used for general-purpose computing and we give some successful examples for each research area. We go into more detail on a paper by Fan et al. [2] to get acquainted with the use of the GPU as general purpose processor. The paper describes a streaming collision detection algorithm between star shaped objects that is mapped to a stream processor.

Finally we made some implementations that use the GPU to speed up computations. We discuss our implementation of a particle system that runs on the GPU, give an overview of the algorithm and discuss the results we achieved. We also discuss our implementation of a math-library on the GPU. The library supports various vector and matrix operations that are performed on the GPU by storing the data in textures. As a conclusion we compare the speed of the GPU implementation with CPU based methods.

Acknowledgments

I would like to thank my promoter Prof. dr. Frank Van Reeth and my supervisor Tom Van Laerhoven for giving me the chance to work on this subject and for their valuable assistance.

Many thanks to the *Expertise Centrum voor Digitale Media* where I did my summer intern which proved to be a good preparation for my thesis.

Also, I would like to thank Tony Witters for correcting all my spelling mistakes and Yannick Francken for the interesting discussions we had about the fragment processor.

Finally I would like to thank my parents for supporting me, and my sister for a final spelling check.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Almost every modern computer gets shipped with a high quality video card that contains a flexible and powerful processor. The purpose of these cards is to generate realistic realtime graphics in today's video games. Under the influence of the multi-billion dollar games industry that continues to develop games that push the latest generations of video cards to their limits, the power of GPUs increases much faster than the power of CPUs [6]. Because of this trend a new research area has arisen that tries to make use of the GPU with it's massive computational abilities as a coprocessor to solve more general problems: General-Purpose Computing on GPUs.

In this thesis, we take a closer look at the graphics processor to see how it has evolved over the last couple of years and why it is so powerful. We also take a look at how the GPU can be used to perform general purpose tasks by discussing the available operations on current GPUs and how they compare to CPU operations. We discuss some of the research areas in which the GPU has been successfully used to perform general purpose tasks and take a closer look at one such example. Finally we have created some implementations that make use of the GPU to speed up several tasks that would otherwise be performed by the CPU.

Chapter 2

The Graphics Processing Unit

2.1 History

In this section we give an overview of the history of graphics cards. They can be subdivided into a number of generations.

Let's first explain the difference between a Graphics Processing Unit (GPU) and a Graphics accelerator. Six years ago, computer workstations contained graphics accelerators, as the name implies they only accelerated graphics [8]. If you removed the graphics accelerator, the computer would perform exactly the same rendering operations, only more slowly. With the arrival of GPUs the concept of graphics acceleration advanced to graphics processing. GPUs became programmable and changes were made to the graphics pipeline, which had nearly been unchanged for the past twenty years. The pipeline was broken down to its basic components and was rebuilt out of programmable, parallel-pipelined processors.

A rough version of the graphics pipeline is shown in figure 2.1. The pipeline starts with the application stage which consists of the program running on the CPU that is feeding commands to the graphics subsystem [4]. In the geometry stage the vertex positions that were supplied by the application are transformed from their 3D position to their position on the screen. The geometry stage also computes extra attributes like vertex colours (influenced by the lighting) or texture coordinates. The resulting 2D triangles will be used as input for the rasterization stage. In this stage per-pixel operations are performed, each triangle will be rasterized (converted to pixels) and vertex attributes will be interpolated across the triangle. The pixels will then be shaded and in the last step the visibility of the pixels is resolved by z-buffering. The resulting pixels will be sent to the framebuffer.

As opposed to graphics accelerators the geometry and the rasterization stage has become completely programmable on GPUs. This allows developers to write programs (shaders) that are executed on the GPU. This way, the GPU will help to reduce the computational burden on the Central Processing Unit (CPU).

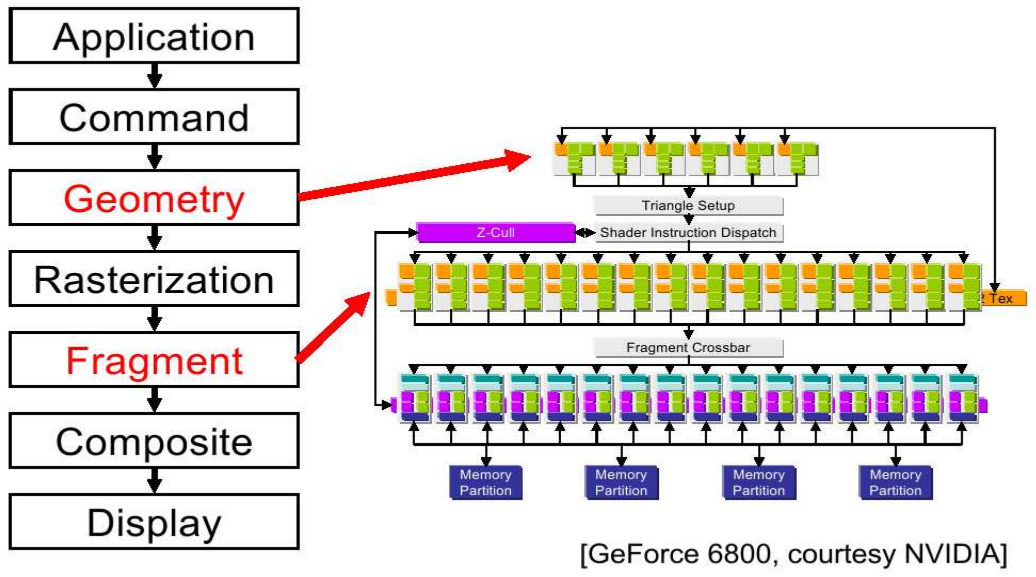


Figure 2.1: The graphics pipeline [3]

In the mid-1990s, the world's fastest graphics hardware consisted of multiple chips that worked together to render images and to output them to the display. By the improvement of semiconductor technology over the years hardware engineers were able to incorporate the functionality of complicated multi-chip designs into a single graphics chip. Current graphics chips have more transistors present in each microchip than current CPUs. For example, the NVIDIA GeForce 6800 contains 222 million transistors while the Pentium 4 Prescott 3.2Ghz contains 125 million transistors.

Industry observers have identified four generations of GPU evolution so far, with each generation giving better performance and a richer feature set for programmability [4].

2.1.1 Pre-GPU Graphics Acceleration

Before the introduction of GPUs, there were companies like Silicon Graphics (SGI) and Evans & Sutherland (E&S) [4] that designed specialized graphics hardware. These systems played an important role in the historical development of computer graphics but because they were so expensive they never achieved the mass-market success of single-chip GPUs. Nevertheless, these systems introduced the first hardware-based solutions for vertex transformation and texture mapping.

2.1.2 First-Generation GPUs

The first generation of consumer-level graphics processors appeared around 1998. They include the NVIDIA TNT2, the ATI Rage and the 3DFX Voodoo3 [4]. These cards are capable of rasterizing pre-transformed triangles and multi-texturing which allowed blending of two textures in the rasterization step. The set of blending functions available to the programmer was very limited. These processors had around 10 million transistors [9].

2.1.3 Second-Generation GPUs

The second generation of GPUs appeared late 1999, early 2000. They include the NVIDIA GeForce 256 and GeForce2, ATI's Radeon 7500 and S3's Savage3D [4]. These cards are also called the T&L GPUs because they off-load 3D vertex transformation and lighting from the CPU. The vertices can be passed untransformed because the GPU is responsible for keeping the transformation matrices and making the transformation and lighting calculations. This is a great improvement for real-time graphics applications because these operations are performed constantly while rendering a scene. Although the available operations for the programmer to combine textures and colour pixels have expanded in this generation of GPUs, the possibilities are still limited. This generation is more configurable, but still not truly programmable. These GPUs contained about 25 million transistors [9].

2.1.4 Third-Generation GPUs

The third generation of GPUs arrived in 2001. They include NVIDIA's GeForce3 and GeForce4 Ti and the ATI Radeon 8500 [4]. This is the first generation that provides vertex programmability rather than merely offering more configurability. Vertex programs have a maximum of 128 instructions. Fragment programs are limited by the way they can access texture data (dependant lookups are limited) and the format of the texture data (only fixed point data). The programs are also limited by the fact that there was no program flow control (branching is impossible). The first attempts of general purpose computing were performed on these GPUs. They contain around 60 million transistors [9].

2.1.5 Fourth-Generation GPUs

The fourth generation of GPUs date from 2002 and on, they include the NVIDIA Geforce FX and GeForce 6800 series and ATI Radeon 9700 9800 and X800 series [4]. Vertex and fragment shaders can now have thousands of instructions. The latest cards even support infinite length vertex and fragment shaders. This level of programmability enables developers to move complex vertex transformations and pixel-shading operations from the CPU to the GPU. These cards also have floating-point units, which makes them more interesting for general purpose tasks that require greater precision. The latest GPUs contain more than 200 million transistors [9].

Generation	Year	Product	Process	Transist.	AA Fill Rate	Polygon Rate
First	Late 1998	RIVA TNT	0.25 μ	7 M	*50 M*	6 M
First	Early 1999	RIVA TNT2	0.22 μ	9 M	*75 M*	9 M
Second	Late 1999	GeForce 256	0.22 μ	23 M	*120 M*	15 M
Second	Early 2000	GeForce2	0.18 μ	25 M	*200 M*	25 M
Third	Early 2001	GeForce3	0.15 μ	57 M	*800 M*	30 M
Third	Early 2002	GeForce4 Ti	0.15 μ	63 M	1200 M	60 M
Fourth	Early 2003	GeForce FX	0.13 μ	125 M	2000 M	200 M
Fourth	Early 2004	GeForce6	0.13 μ	220 M	4000 M	375 M

Table 2.1: Feature overview [4]

Table 2.1 lists an overview of the features and it uses a couple of terms :

Process: the minimum feature size in microns (μ, millionths of a meter) for the semiconductor process used to fabricate each microchip

Transistors: an approximate measure, in millions (M), of the chips' design and manufacturing complexity

Antialiasing fill rate: a GPU's ability to fill pixels, measured in millions (M) of 32-bit RGBA pixels per second, assuming two-sample antialiasing, numbers in *italics* indicate fill rates that are de-rated because the hardware lacks true antialiased rendering

Polygon rate: a GPU's ability to draw triangles, measured in millions (M) of triangles per second

2.2 Motivation

In this section we will explain why the use of Graphics Hardware to perform general-purpose computing has become so popular during the last couple of years.

2.2.1 Computational Power

The latest generations of GPUs are extremely fast, table 2.2 gives an overview of the number of GFLOPS (billion floating point operations per second) of modern hardware [6]. You immediately notice that graphics processors easily outperform regular CPUs. The GeForce 6800 Ultra can even perform more than four times as many floating point operations per second than a Pentium 4 3GHz.

Processor	GFLOPS
Pentium 4 3GHz	12 (theoretical)
GeForce FX 5900	40 (observed)
GeForce 6800 Ultra	53 (observed)
Radeon X800	42 (observed)

Table 2.2: GFLOPS overview [6]

Another aspect of why GPUs are so fast is memory bandwidth [10]. Table 2.3 gives an overview of the memory bandwidth in different parts of a computer system. As you can see the GPU has a great amount of internal bandwidth at its disposal. This bandwidth can be used to get dramatic performance improvements compared to the CPU. The communication between the CPU and the GPU happens through the PCI-express bus or AGP-bus on the motherboard. PCI-express is the new standard that was introduced in 2004 and is the successor to AGP. The biggest improvement of PCI-express is not just the increased speed but the fact that data transfers are equally fast in both directions (4 GB/sec in every direction) while readback of the AGP-bus happens at PCI-speed (∼256MB/sec). This is especially useful for general-purpose computing because the results of the computations often need to be transferred back to main memory.

Component	Bandwidth
GPU memory interface	35 GB/sec
AGP Bus (x4)	1066 MB/sec
PCI Express Bus (x16)	8 GB/sec
CPU memory interface (800 MHz Frontside bus)	6 GB/sec

Table 2.3: Memory bandwidth overview [4]

Add to these facts that GPU speed is increasing faster than Moore's law [6] and you will realise the GPU is a very interesting platform to perform heavy computations.

2.2.2 Parallelism

CPUs normally have only one processor. GPUs on the other hand have two types of programmable processors: the vertex processor and the fragment processor [10]. Each processor operates in a different stage of the graphics pipeline but both processors allow parallel data-processing. Modern GPUs have 6 vertex pipelines en 16 fragment pipelines.

GPUs have SIMD (Single Instruction Multiple Data) characteristics [11]. They have native support for 4-tuples (positions, colours) which allows operations on vectors at the cost of 1 instruction. Another SIMD characteristic of GPUs is stream processing. The same function (shader) is applied to each element in a stream, such a function is also called a kernel. The use of streams allows a great level of data parallelism and because there are very few dependencies between elements in a stream, it encourages high arithmetical intensity [6].

2.2.3 Flexibility and Precision

GPUs have programmable pixel and vertex engines. Thanks to high level shading languages developers gain much more flexibility when writing vertex and fragment shaders. Before the introduction of these languages, shaders were written in assembly languages provided by vendors but they had portability issues and were tiresome to code. Thanks to the high level languages, writing shaders has become much more intuitive which allows easier shader creation, easier code reuse and easier debugging. Next we will give an overview of the most frequently used shading languages.

C for Graphics (Cg)

Cg is the shading language developed by NVIDIA. Cg has a similar syntax to ANSI C, but it also adopts some ideas from modern languages such as C++ and java and from earlier shading languages like Renderman or Stanford Shading Language [4]. A schematic view of the Cg model is shown in figure 2.2. Cg is platform independant thanks to its compatibility with OpenGL. Cg is also compatible with Direct3D. To cope with the divergence of available graphics hardware the Cg API introduces profiles [12]. A Cg profile supports a subset of the Cg Language, this way a program written against a certain profile is guaranteed to run on all hardware that supports the features of that profile.

High Level Shading Language (HLSL)

HLSL was introduced with the release of DirectX 9. Shaders were first supported in DirectX 8 but had to be written in assembly back then.
Cg and HLSL are actually the same language. Cg/HLSL was co-developed by NVIDIA and Microsoft. They have different names for branding purposes.

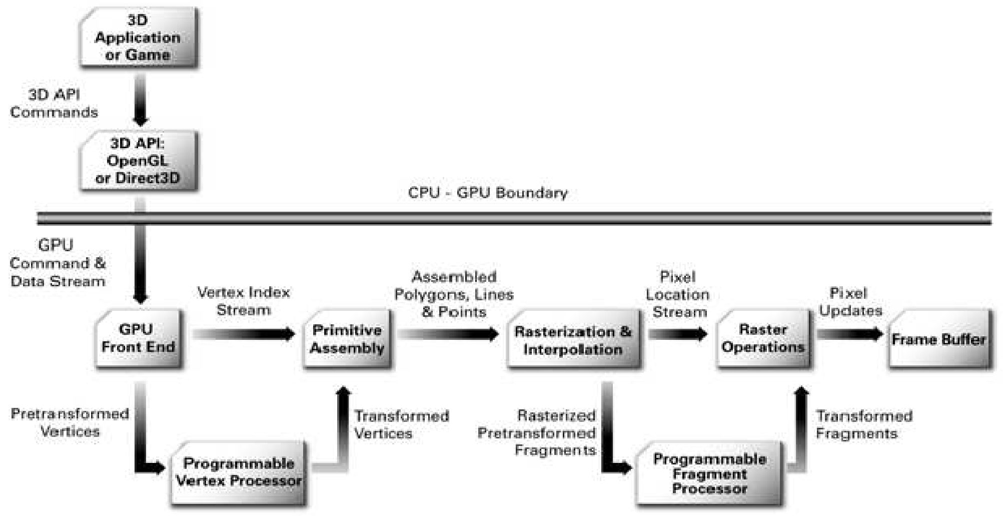


Figure 2.2: The Cg Graphics pipeline [4]

HLSL is part of Microsoft's DirectX API and only compiles into Direct3D code, while Cg can compile to DirectX and OpenGL.

OpenGL Shading Language (GLSlang)

GLSlang is available since OpenGL 1.4 and is a part of the core OpenGL 2.0 specification [13]. The OpenGL shading language is based on ANSI C and many of the features have been retained except when they are in conflict with performance or ease of implementation. C has been extended with vector and matrix types (with hardware based qualifiers) to make it more concise for the typical operations carried out in 3D graphics. Some mechanisms from C++ have also been borrowed, such as overloading functions based on argument types, and the ability to declare variables where they are first needed instead of at the beginning of blocks.

GPUs also gained a lot of precision over the last years. The latest models of NVIDIA support 32-bit floating point (fp32) precision over the entire pipeline [10] which allows general-purpose algorithms to return accurate results. ATI currently supports 24-bit precision. Hower fp32 does not mean fully IEEE 754 compliant [14], the results can show slight deviations.

2.3 Limitations

Programming on the GPU also introduces some limitations. GPUs are not designed to perform general purpose tasks, they are designed to provide realistic realtime graphics for computer games or 3D visualisations. This causes some limitations [6]

You have to use a graphics API: this is necessary if you want to execute shaders. As a result algorithms have to be converted to a format that is supported by that API. For example, if you want to pass data to a shader you can not just put it in an array and pass it as an argument because this is not supported. The only thing that can be passed as arguments are vectors and textures. So all data will have to be stored in textures to be able to access it from a shader.

You have a limited memory interface: the data you use for computations is stored in textures This causes some limitations because texture access in a shader is read-only. As a result, operations on fragments in one rendering pass need to be completely independent of each other because the kernels are executed in parallel.

Lack of debuggers and profilers: for a long time the only possibility to debug your shaders was to read back the texture from the graphics card memory and investigate the colour values. However support for profiling and debugging is gradually improving with recently released tools like gDEBugger or NVShaderPerf.

Limited bandwidth from the GPU to the CPU: currently the most used interface to communicate with the graphics card is still the AGP-bus of which readback is limited to PCI-speeds (∼256MB/sec). This should be resolved in the near future with the growing number of PCI-express cards.

2.4 Future

In the previous section we have seen that graphics hardware has gone through some great improvements over the last couple of years, not only in performance but also in the amount of available features. In this section we give an overview of what to expect from future hardware.

2.4.1 Increased Flexibility

New features are constantly being added to the graphics APIs which are used to communicate with the GPU. These new features will help developers to access the graphics hardware more efficiently and in an easier way. High level shading languages also keep evolving by providing instructions to access new hardware features when they become available and with better optimization techniques to convert the shaders into more efficient assembly code.

As a consequence of these improvements more algorithms will become suitable to be converted to the GPU. This will allow the GPU to be used in a growing number of research domains.

2.4.2 Easier Programming

In the future it might be possible to perform computation on the GPU with non-graphics APIs and languages [15]. This would be a big improvement for developers of general-purpose algorithms on the GPU because they would not have to create textures and other graphics-specific objects to store their data.

A first step in the right direction is the BrookGPU shading language. This language provides an abstraction to the graphics API. Programs written in BrookGPU will be converted to instructions of the graphics API which is very useful for developers that have little experience with graphics APIs but want to convert their algorithms to the GPU.

2.4.3 Increased Performance

GPUs will keep getting faster. The major driving force behind the graphics industry is the computer games industry. As computer games keep getting more complex they need faster GPUs. The cards will be equipped with more vertex and fragment processors and will have better branching techniques [15] which will allow more complex shaders.

As a consequence of the increased power, GPUs will be able to produce more and more GFLOPS. An expected performance trend is shown in figure 2.3. As you can see we are rapidly approaching the point where GPUs will be able to perform TFLOPS. The GPU will ultimately encapsulate the power of a supercomputer on a single chip!

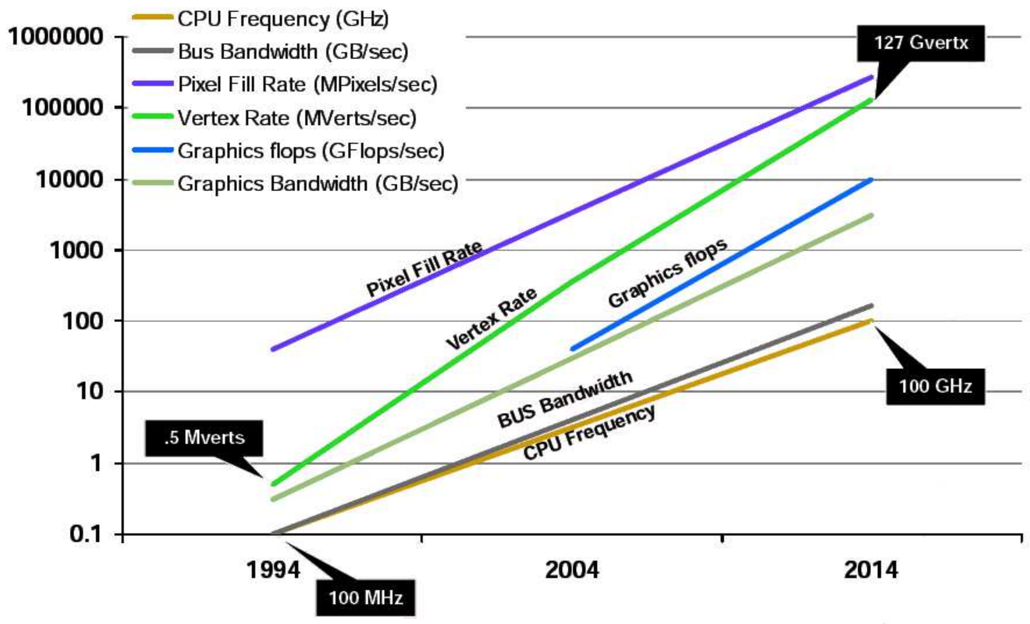


Figure 2.3: Trends in GPU evolution [5]

2.4.4 New Hardware Features

Note that this section discusses future features and technologies (either developing or developed) that are based on unofficial information and are expected to be included in shader model 4.0 [16, 17].

A few possible new features are:

Geometry shader: this will be a new type of shader that is responsible for creating new vertices. This will be an addition to the currently available vertex shader which is only capable of transforming vertices.

Unified Shader Model: this implies that fragment and vertex shaders will be identical in both syntax and feature set. This will allow the hardware to combine the different hardware units together in a single pool. This has an added benefit that any increase in shading power increases both vertex and pixel shading performance. This would also mean that these unified shaders would always be 100% loaded and would make it easier for programmers because they would not need to waste time searching for bottlenecks.

CPU-like data types: GPUs will have support for data types as used in CPU-programming (e.g. integer)

IEEE 754 compliant floating points: currently GPUs have support for 32-bit floating point values but these are not full IEEE 754 compliant [14]. This could change in the next shader model. This would finally allow GPU algorithms to run in 32-bit precision with as much accuracy as the algorithms on the CPU in 32-bit precision.

Chapter 3

Programming the GPU

In this chapter we explain the basics of programming the GPU. We discuss the stream programming model and give an overview of the operations that are available and unavailable on the GPU. Finally we discuss some analogies between programming on the GPU and on the CPU.

3.1 The Stream Programming Model

The concept of viewing the graphics hardware as a streaming processor was first proposed by Purcell et al. [18]. Streaming computing differs from traditional computing in that the system reads the data required for a computation as a sequential stream of elements. Each element of a stream is a record of data requiring a similar computation. The system executes a program or kernel on each element of the input stream placing the result on an output stream. In this sense, a programmable graphics processor executes a vertex program on a stream of vertices, and a fragment program on a stream of fragments [18].

The streaming model leads to efficient implementations for three reasons [18]:

1. Because each element in a stream of data is independent from the other elements, additional pipelines can be added to the hardware to process elements of the stream in parallel.

2. Kernels achieve high arithmetic intensity, a lot of computation is performed per fix-sized record. This results in a high computation to memory bandwidth ratio.
3. Streaming hardware can hide the latency of texture lookups by using prefetching. When the hardware fetches a texture for a fragment, the fragment registers are placed in a FIFO and the fragment processor starts processing another fragment. Only after the texture has been fetched does the processor return to that fragment.

General-purpose computing on GPUs often relies on the stream programming model. Textures represent the streams of data and the kernel is the fragment program. The fragment program is applied to each pixel of the texture.

3.2 Operations on the GPU

The operations that are available on the GPU are not as familiar as the ones on the GPU, they are more graphics-centric. In this section we will give an overview of the operations that are available to work with on the GPU.

3.2.1 Available Operations

The available operations on the GPU are [19]:

Read-only memory: The GPU has access to read-only memory. This is done by performing texture lookups.

Random access read-only memory (gather): It is also possible to perform random access to the texture by generating texture coordinates inside the fragment shader. This allows the gather operation to be implemented in fragment shaders. A gather operation is an operation that computes a value in a grid based on other values of the grid (cf. figure 3.1).

Per-data-element interpolants: Arguments can be given to the fragment shader that vary per data element and are interpolations between values. An example is the texture coordinate that will be used to fetch a value from the texture. This value will be interpolated according to the values that were generated in a vertex shader or that were specified by the programmer with commands supplied by the graphics API. These values will be stored in varying registers.

Temporary storage: Temporary values can be created in shaders. These values can be used in computations and will be stored in local registers.

Read-only constants: Constants can be passed to a shader after the creation. These values will then be used every time the shader is executed but can not be altered. They are stored in constant registers.

Write-only memory: The output values of a shader are normally written to the framebuffer. But with the render-to-texture mechanism these values can be rendered directly to a texture. So these textures function as write-only memory because they can not be bound as an input texture at the same time.

Floating point ALUops: Shaders support arithmetic operations like addition, subtraction, multiplication and division on floating point values.

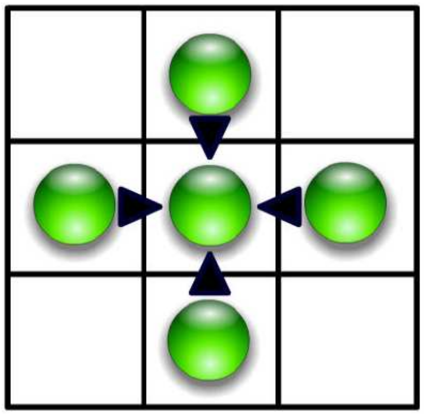


Figure 3.1: The gather operation [6]

3.2.2 Missing Operations

Some operations are missing on the GPU. Reasons are that there is a lack of demand for these operations from game developers and some operations do not scale well to the stream programming model:

- Stack
- Heap
- Integer or bitwise operations
- Scatter operation: a scatter operation requires nearby elements in a grid to be updated, based on computations involving the current element of the stream (cf. figure 3.2). This is not possible on a GPU because a fragment program can only output a value to the address of the current pixel.
- No reduction operations (min, max, sum): it is not possible to perform operations based on all the elements in the stream inside a shader.
- Limited number of outputs: currently fragment shaders can only output four values per pixel.

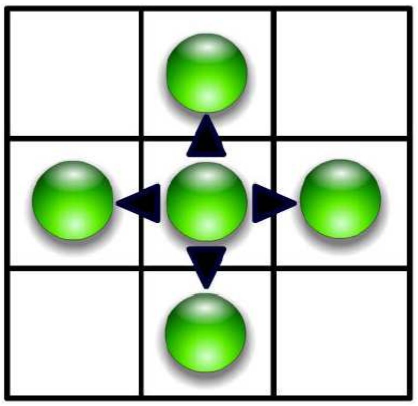


Figure 3.2: The scatter operation [6]

3.2.3 Handling Some Missing Operations

In this section we will give a couple of examples of how some missing operations can be performed on the GPU.

Scatter Operation

There are two possible solutions to perform a scatter operation. A first solution would be to try to convert the scatter operation into a gather operation, since gather operations work well in a fragment program.

Another solution would be to actually emulate the scatter operation with the help of the vertex processor. A vertex program can change the position of the current vertex but cannot read from other vertices. By rendering points instead of quads this enables the shader to perform a scatter operation. The position of the vertex will be changed to the position of the value that needs to be adjusted. These values can then be used again as input to the vertex shader by using the render to vertex array method. This method is described in section 6.3.5

Reduction Operation

The reduction operation (min, max, …) can be emulated by a fragment program. This is done by repeatedly performing gather operations until only a single value is left. A more detailed description of this mechanism can be found in section 7.3.4. The problem with this emulation is that multiple render passes are needed to perform the reduction, which can be costly.

3.3 CPU - GPU Analogies

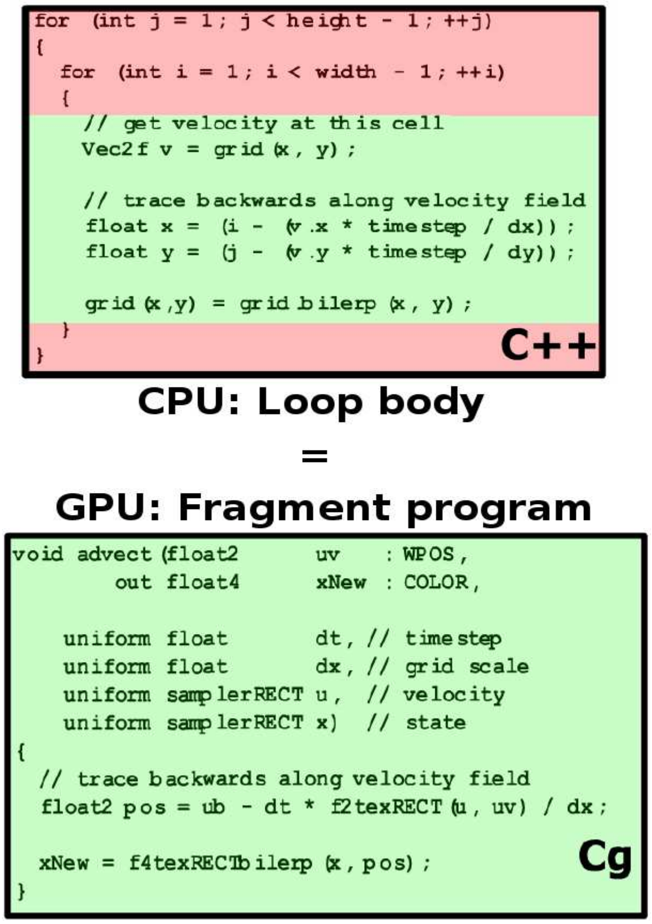
In this section we will show some anologies between programming on the CPU and the GPU [6].

3.3.1 Data-lookups

Retrieving data on the CPU is done by reading data out of the main memory, the data is retrieved by providing the address of the data in the memory. On the GPU data is retrieved by performing a texture lookup. The data is retrieved by providing the coordinate of the pixel that contains the data.

3.3.2 Looping

When a particular action needs to be performed on lots of elements, a loop is created on the CPU with a `for` or `while` statement. On the GPU such an action is implemented in a fragment shader. The shader will be executed for every pixel in a texture. When converting algorithms from the CPU to the GPU the body of a loop will often end up in a shader as shown in figure 3.3.



```
for (int j = 1; j < height - 1; ++j)
{
  for (int i = 1; i < width - 1; ++i)
  {
    // get velocity at this cell
    Vec2f v = grid(x, y);

    // trace backwards along velocity field
    float x = (i - (v.x * timestep / dx));
    float y = (j - (v.y * timestep / dy));

    grid(x,y) = grid.bilerp(x, y);
  }
}
```

```
void advect(float2        uv    : WPOS,
        out float4        xNew  : COLOR,

    uniform float         dt,  // timestep
    uniform float         dx,  // grid scale
    uniform samplerRECT   u,   // velocity
    uniform samplerRECT   x)   // state
{
  // trace backwards along velocity field
  float2 pos = ub - dt * f2texRECT(u, uv) / dx;

  xNew = f4texRECTbilerp(x, pos);
}
```

Figure 3.3: A loop on the CPU and the GPU [6]

3.3.3 Saving Data

Saving data on the CPU simply implies writing the data to an element of an array. Saving data on the GPU is done with the render-to-texture mechanism. Instead of writing the output values of the shader to the framebuffer the values are written to a texture that is bound to an off-screen framebuffer. This does mean that double-buffering techniques sometimes have to be used when the output texture will be used as the input texture to the same shader in the next pass because you can not use the same texture as input and output of a fragment program. The use of the double-buffering method and the needed extension is explained in more detail in section 6.4.1.

3.3.4 Algorithm Step

On the CPU a step of an algorithm usually resides in a method that contains a loop where the algorithm is executed. Invoking that method will cause the algorithm to advance a step.

Computation on the GPU is invoked by drawing a pixel. This will cause the fragment shader to be executed. To perform an algorithm step, all the values need to be updated by a shader. Invoking computation on all the values is done by drawing geometry. The most common operation in GPGPU is to draw a viewport sized quad. This will cause all the values of the framebuffer to be updated. Setting up the viewport is shown in figure 3.4. After the viewport has been setup, the correct framebuffer needs to be activated that will be used as a render target. Then the input textures of the fragment shaders are assigned and finally a viewport sized quad is rendered that will cause all the values to be updated.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2d(0, 1, 0, 1);
glViewport(0, 0, resX, resY);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

Figure 3.4: Setup viewport in OpenGL

Chapter 4

Research Areas

In this chapter we give an overview of some research areas in which general-purpose computing on graphics hardware has been successful.

4.1 Databases

GPUs have been successfully used as a coprocessor to accelerate various database operations.

Lloyd et al. [20] describe some algorithms for performing fast computation of several database operations on graphics hardware. In particular the paper describes operations such as conjunctive selections, aggregations and semi-linear queries which are heavily used in typical database, data warehousing and datamining applications. The paper compares the performance of these operations with an optimized CPU-based implementation. The experiments indicate that the graphics processor is an effective coprocessor for performing database operations.

Sun et al. [21] discuss how the GPU can be effectively used to accelerate the performance of spatial database operations. Spatial database operations, especially which involve polygon datasets, have been known to be computationally expensive. The paper describes a hardware / software coprocessing technique which uses basic features of a GPU to reduce the spatial query processing cost. Experimental evaluation shows that their hardware-based approach can significantly outperform leading software-based techniques. However, this evaluation is done in a stand-alone setting where there are no indices, preprocessing or other optimizations available in a database.

4.2 Audio and Signal Processing

GPUs have also been used to perform audio effects or calculate audio accoustics.

Jędrzejewski [22] proposes a way to accelerate the computation of sound paths between sound sources and receivers by using the GPU. The algorithm is similar to a raytracer, rays are cast from audio sources. If they intersect with the sphere, representing an approximation of the user, after a number of steps they will be included in an echogram that is used in the auralization process. The application can run in real time with up to 64.000 rays.

BionicFX [23] created a technology for music production that turns NVIDIA video cards into audio effects processors. It uses a mechanism called Audio Video Exchange (AVEX) to convert digital audio into graphics data, and then perform effect calculations using the GPU. The AVEX technology allows music hobbyists and professional artists to run high quality audio effects at high sample rates on their desktop computer.

4.3 Advanced Rendering

A lot of research has been done to perform advanced rendering techniques like ray tracing, global illumination and image based modeling and rendering on the GPU.

Purcell [24] has done research on GPU-based ray tracing and photon mapping algorithms. His results prove that advanced rendering techniques can be successfully run on modern graphics hardware.

Hillesland et al. [25] have created a framework used to solve two distinct image-based modeling problems: light field mapping approximation and fitting the Lafortune model to spatial bidirectional reflectance functions. The graphics hardware implementation outperformed the CPU implementation with a 5-fold speedup.

4.4 Computational Geometry

Another research area that has found the GPU useful to perform some heavy tasks is computational geometry (eg. collision detection, Constructive Solid Geometry (CSG)).

Stewart et al. [26] have created an algorithm for Overlap Graph subtraction Sequences on the GPU and describe how it can be combined with the Sequenced Convex Subtraction (SCS) algorithm for Constructive Solid Geometry. An overlap graph stores the spatial relationship of the objects in a CSG product. Nodes in the graph correspond to shapes or objects while edges in the graph indicate spatial overlaps. Experimental results indicated speed-up factors of up to three.

Pascucci [27] has developed a technique to compute isosurfaces on programmable GPUs. Using the vertex programming capability of modern graphics cards the cost of computing an isosurface from the CPU is transfered to the GPU. This has the advantage that the task is off-loaded from the CPU and it can be avoided to store the surface in main memory.

4.5 Image and Volume Processing

GPUs have been successfully used to perform image and volume processing tasks.

Moreland et al. [28] describe an implementation of the Fast Fourier Transform (FFT) on the GPU. The Fourier transform is a well known and widely used tool in many scientific and engineering fields. It is essential for many image processing techniques, including filtering, manipulation, correction and compression. The computer graphics community could benefit greatly from such a tool if it were part of the graphics pipeline.

Krueger et al. [29] have done research for acceleration techniques for GPU-based Volume Rendering. Direct volume rendering via 3D textures is a useful tool for the display and visual analysis of volumetric scalar fields. The paper proposes some acceleration techniques like early ray termination and empty space-skipping resulting in performance gains of factors up to 3.

4.6 Scientific Computing

Another research area that benefits from high performance GPUs is scientific computing. GPUs can be used to perform lots of arithmetic computations allowing more realistic simulations in real-time.

Fan et al. [30] have created a GPU cluster for high performance scientific computing. As an example application, they have developed a parallel flow simulation using the Lattice Boltzmann Model (LBM) on a GPU cluster and have simulated the dispersion of airborne contaminants in the Times Square area of New York City. With 30 GPU nodes they achieved speeds that were 4.6 times faster than their implementation on a CPU cluster.

Liu et al. [31] present a way to process complex boundary conditions when simulating fluid flow using the Navier-Stokes Equations on the GPU. The test results prove the efficiency of the method, and as a result, it is feasible to run middle-scale problems of 3D fluid dynamics at interactive speeds with complex geometry.

Chapter 5

In-depth Example: Collision Detection

5.1 Introduction

Real-time collision detection is a classic and important problem in areas like computer graphics, virtual reality, computer games, CAD, robotics and manufacturing. Recently the GPU has been used to speed up the process of collision detection. Fan et al. [2] describe an implementation on the GPU with the goal of maximum performance at the lowest cost. The implementation performs collision detection between star shaped objects and is mapped to the streaming processor by performing lots of ray triangle intersections.

5.2 Algorithm

5.2.1 Overview

The goal is to perform collision detection between star-shaped objects. A star-shaped object is defined as an object in which there exists at least one point O (we call O the origin of the object) and any semi-finite ray originating from the point O will intersect the surface of the object at exactly one point [2]

Mapping collision detection to programmable GPUs is done in three steps: ray generation, ray triangle intersection and the comparison of the threshold values as indicated in figure 5.1.

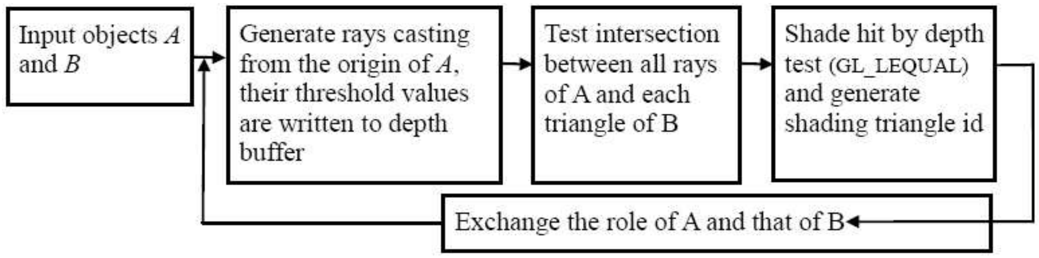

Input objects A and B
Generate rays casting from the origin of A, their threshold values are written to depth buffer
Test intersection between all rays of A and each triangle of B
Shade hit by depth test (GL_LEQUAL) and generate shading triangle id
Exchange the role of A and that of B

Figure 5.1: Algorithm overview

5.2.2 Ray Generation

In the first step of the algorithm rays are emitted from the origin of an object to its surface vertices. Each ray is represented parametrically as shown in equation 5.1, where *orig* is the ray origin and *dir* is the ray direction.

$$ray(t) = orig + t \cdot dir \tag{5.1}$$

For each ray, the distance between the origin and the surface vertices is calculated and recorded (this distance is called the threshold value). Since all the rays are emitted from the same origin, it can be stored as a constant vector. The ray directions and corresponding thresholds are stored in an RGBA texture. The red, green and blue channels are used to store the direction and the alpha channel is used to store the threshold value. So each ray corresponds to a pixel in the texture. The texture size is determined by the number of rays which in turn is determined by the number of vertices of the object. The way to determine the size of the texture is given in equation 5.2, where n_r is the number of rays and *width* and *height* denote the size of the texture. Before each step, the threshold values of the rays are stored in the depth buffer and the colour buffer

$$width = height = \lfloor \sqrt{n_r} \rfloor + 1 \tag{5.2}$$

An example of a ray texture is shown in figure 5.2.

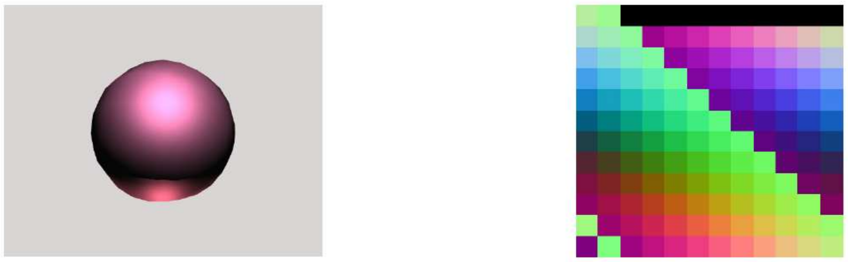


Figure 5.2: A sphere and its corresponding ray texture [2]

5.2.3 Ray - Triangle Intersection

Suppose you have two star-shaped objects: A and B. In each iteration of the ray-triangle intersection, an intersection test will be performed with every ray of object A and one triangle of object B. So multiple passes will be needed to perform intersection tests with every triangle of object B.

To pass the triangle to the fragment shader a quadrilateral is rendered of which the attributes of its four vertices consist of the attributes of the current triangle. The attributes are passed to a vertex shader. Rasterization of this quadrilateral causes the attributes to be interpolated for each pixel of its screen projection. The attributes are the same for each vertex of the quadrilateral and consists of the triangle id which is stored in the colour attribute, the three vertices of the triangle which are stored as texture coordinates and the triangles front facing normal which is stored in a second colour attribute. The vectors v_1, v_2, v_3, n, $e_{12}(= v_2 - v_1)$ and $e_{13}(= v_3 - v_1)$ can be obtained from the vertex shader. v_1, v_2 and v_3 are passed to the vertex shader and the other values are calculated inside the vertex shader and can then be passed to the fragment shader.

Next, the intersection test is performed in a fragment program which executes by rendering a viewport sized quadrilateral. The input data to the ray-triangle intersection shader will consist of two parts: the ray texture and the triangle data. This is shown in figure 5.3

The intersection test is shown in figure 5.4. The test passes if all three barycentric coordinates u, v and w are all in the interval (0,1). If the ray does not intersect the triangles, the maximum limited value is returned as its parameter t. A depth test function is used to test parameter t.

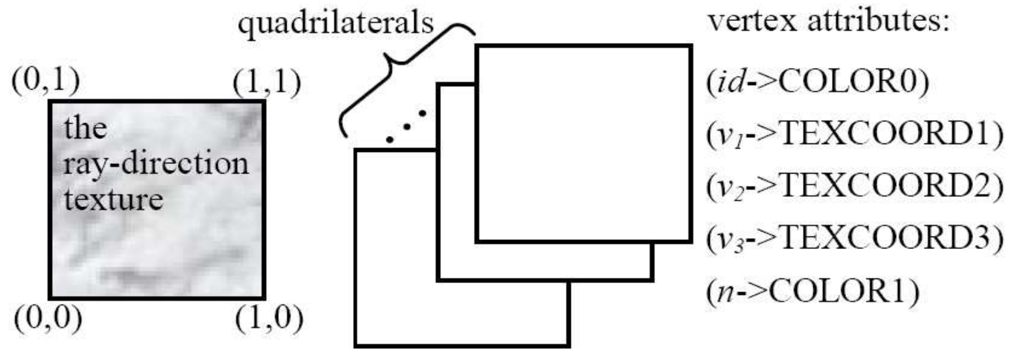


Figure 5.3: The stream of input data [2]

```
float4 Detect_Rays_with_Triangle(   float3 orig, float3 dir, float3 v1,
                                    float3 v2, float3 e12, float3 e13 )
{
    float3 v1o = orig - v1;                 float3 v2o = orig - v2;
    float3 v1od = cross( v1o, dir);         float3 v2od = cross( v2o, dir );
    float3 ed = cross( e, dir );
    bool bHit = false;                      float det = dot( e13, e12d);
    bHit = (det< -0.000001f) ? true: bHit;  bHit = (det>= 0.000001f) ? true: bHit;
    float u = dot( e13, v1od ) / det;       float v = - dot( e12, v1od ) / det;
    float w = u+v;
    bHit = (u>=0.0f) ? bHit:false;          bHit = (u<1.0f) ? bHit:false;
    bHit = (v>=0.0f) ? bHit:false;          bHit = (w<1.0f) ? bHit:false;
    t = (bHit) ? (- dot( n, v1o)/ dot( n, dir) ) : MaxT;
    return bHit;
}
```

Figure 5.4: Ray-triangle intersection [2]

5.2.4 Compare Threshold Values

After the ray-triangle intersection tests, a stream of ray triangle hits is outputted. It is then determined via depth comparison whether the depth buffer and colour buffer should be updated. At each pixel of the viewport, if the ray-triangle intersection value t is smaller than the current depth value in the depth buffer, the depth buffer is updated with the t value and correspondingly, the colour indexed by the triangle id is written to the colour buffer. If there exists at least one pixel of which the value has been updated to a colour other than white, the two objects overlap.

5.3 Optimizations

Some optimizations can be applied to the presented algorithm to increase performance.

5.3.1 Direction Cone

In order to decrease the number of rays that need to be tested, a direction cone is created. Before the collision detection takes place, the overlapping region of the bounding volumes of the two objects is determined. Next, the bounding sphere of this region is created. Now the direction can be created by setting the ray origin as the cone's apex and by setting the line between the ray origin and the center of the bounding sphere as the cone's axis. This is shown in figure 5.5.

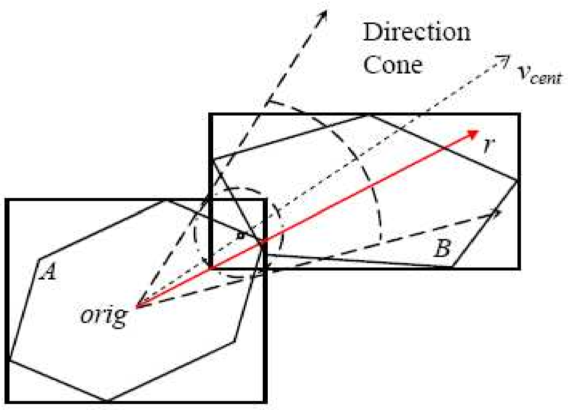


Figure 5.5: The direction cone [2]

All the rays starting in the origin can now be checked to see if they are in the direction cone. If that is not the case, the rays can be rejected. This way only a small subset of the rays need to be checked for ray-triangle intersection.

5.3.2 Back-face Culling

In order to further decrease the number of ray-triangle intersection tests a modified version of back-face culling is applied. The normal of every triangle is compared to the axis of the direction cone. If the normal of the triangle is in the opposite direction of the cone's axis an intersection test does not need to be performed. This is shown in figure 5.6.

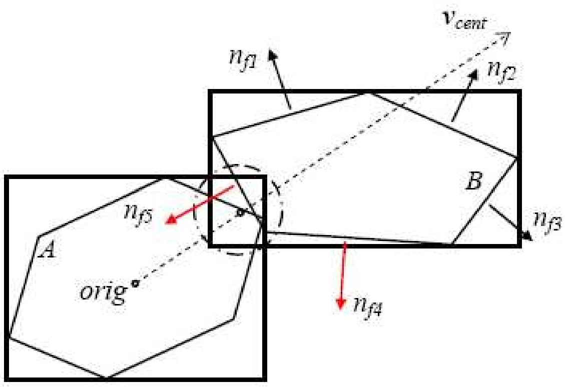


Figure 5.6: Back-face culling [2]

5.3.3 Viewing Volume Culling

OpenGL provides a mechanism to determine which objects of the scene are visible in the current viewport. This is done by switching the rendermode of OpenGL to GL_SELECT with the function `glRenderMode()`. This mode does not necessarily require the display to be updated. By setting the rendermode to GL_SELECT and displaying the scene, OpenGL will return the list of primitives that are located in the viewport.

This method is used to cull triangles that appear outside of the bounding volume of the overlapping region of the two objects. Axis aligned bounding boxes (AABBs) are used as the bounding volumes. The overlapping volume is also an AABB and will be set as the viewing volume. The object will now be rendered with the GL_SELECT rendering mode. The list of triangles that appear in the viewing volume will be returned by OpenGL, the other triangles are culled.

Figure 5.7 shows 2 objects of which the overlapping region, which is set as the viewing volume is marked. The only one of the faces of object B appearing inside of the viewing volume is face f_5. Thus only f_5 needs to be passed to the ray triangle intersection test.

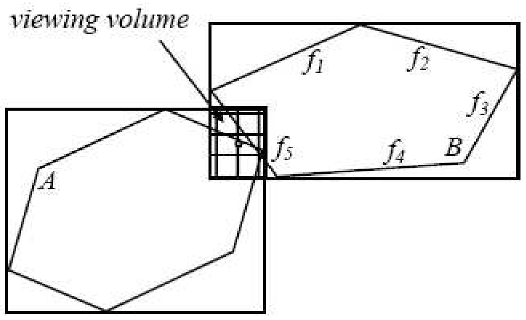


Figure 5.7: Culling by the user-defined viewing volume [2]

5.3.4 Triangles Caching and Organization

The collision detection process can be further optimized by using coherence. When collision is detected, the intersecting triangles' ids are stored in the colour buffer. These values are then read back to the main memory by the CPU and they are put into a triangle cache. In the next frame the algorithm will first render the quads corresponding to the triangles in the cache.

Besides reading back the results in the framebuffer, the CPU is also used to organize the architecture of the algorithm. Figure 5.8 shows a representation of the algorithm.

5.4 Results

The streaming collision detection algorithm was run on a PC with a Pentium 4 1.6 GHz processor with 512MB of RAM and an NVIDIA GeForce4 Ti4200 with 64MB of RAM. The driver is an NVIDIA CineFX (NV30) emulator driver because the graphics card does not support the required NV30 instruction set.

Two scenarios were tested: one with two spheres (530 triangles cf. figure 5.9) and one with two star shaped objects (5420 triangles cf. figure 5.10). Tables 5.1 and 5.2 show that the optimization techniques have been proven to be efficient. The overall performance was not quite satisfactory because emulation drivers had to be used. Moreover, they did not provide a comparison with a pure software implementation of a collision detection algorithm.

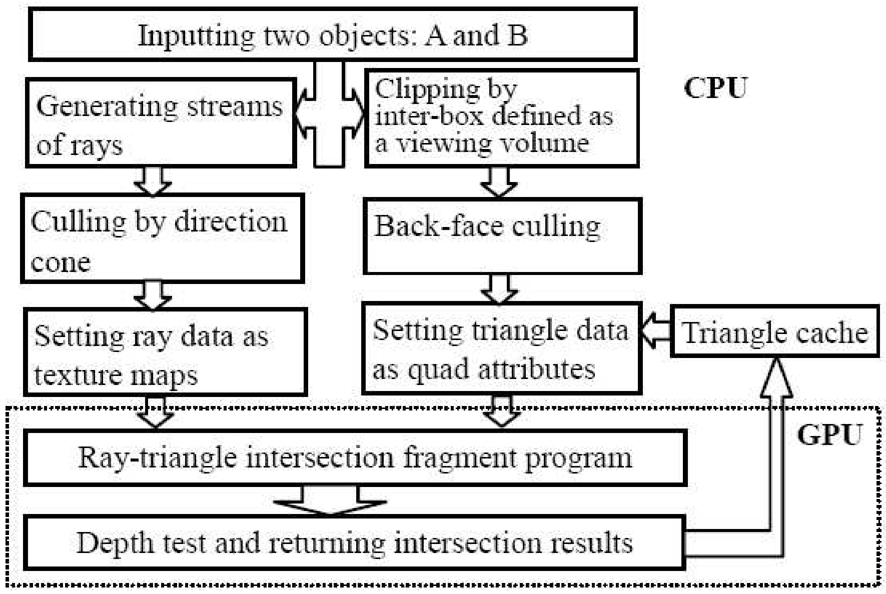

Inputting two objects: A and B
Generating streams of rays
Clipping by inter-box defined as a viewing volume
CPU
Culling by direction cone
Back-face culling
Setting ray data as texture maps
Setting triangle data as quad attributes
Triangle cache
GPU
Ray-triangle intersection fragment program
Depth test and returning intersection results

Figure 5.8: Architecture of streaming collision detection [2]

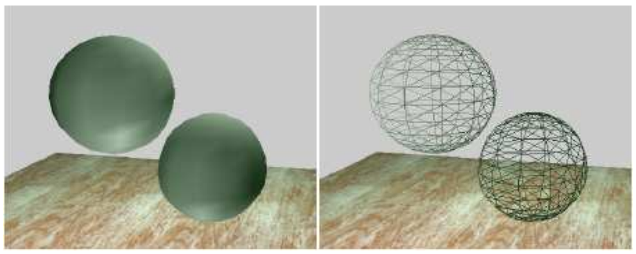


Figure 5.9: Scenario 1: two spheres [2]

We expect that using a hardware NV30 feature set will seriously boost the performance of this implementation making it able to compete with or even outperform software implementations.

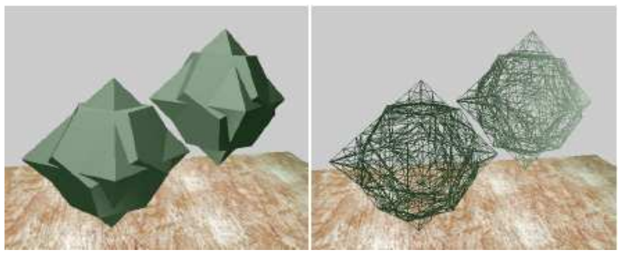


Figure 5.10: Scenario 2: two star-shaped objects [2]

Optimizations	Culling rate (%)	
	Scenario 1	Scenario 2
Direction cone	61.9	78.4
Back-face culling (BFC)	49.6	48.7
Viewing volume culling (VVC)	31.2	36.8

Table 5.1: Culling rates using optimizations [4]

Methods	Scenerio 1		Scenerio 2	
	Tcd (ms)	Speedup	Tcd (ms)	Speedup
No optimizations	191.7	1.00	1129.3	1.00
Direction cone (DC)	132.6	1.45	693.6	1.63
DC + BFC	81.5	2.35	440.2	2.57
DC+BF+VVC+TC	67.2	2.85	330.7	3.41

T_{cd} is the average collision detection time.

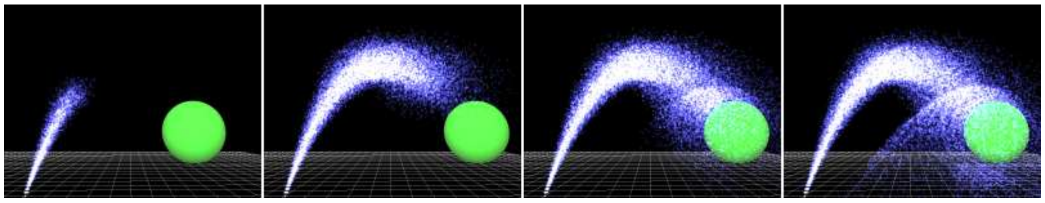
Table 5.2: Collision detection time and speedup [4]

Chapter 6

Particle System

The implementation of the particle system that we made is based on the paper of Lutz Latta: Building a Million Particle System [32]. The paper describes a full GPU implementation of both the simulation and rendering of a particle system. The application renders a particle system in OpenGL while the positions and velocities of the particles are being updated on the GPU by storing the data in textures.

We also made some modifications to the proposed algorithm by creating an implementation that runs entirely on the GPU instead of managing the creation and deletion of particles on the CPU. Furthermore, we used framebuffer objects [33] for rendering to off-screen framebuffers.



6.1 Introduction

Particle systems can be used for pretty much anything you want, however as a general rule, a particle system is a collection of a great number of entities that are either related or unrelated and behave according to a set of logical rules. An example of a particle system is a fountain. The drops of water are the entities and they behave according to the rule of gravity. Entities in a particle system usually have a very limited lifespan and will be replaced by other entities as soon as they die.

Real-time particle systems are usually limited by the fill rate or the communication between the CPU and GPU [32]. The fill rate is the number of pixels the GPU can draw each frame. Fill rate usually is a problem when relatively large particles are used which causes a lot of overlapping between particles. Particle systems generally use small particles because this increases realism so the fill rate limitation loses importance. The second limitation, the transfer of particle data (position, colour, ...) from the simulation on the CPU to the GPU, is the dominating factor. In typical game applications, CPU-based particle systems usually achieve only 10,000 particles per frame while sharing the graphics bus with many other rendering tasks. By moving the simulation of the particles to the GPU, the graphics bus will have more available bandwidth for other rendering tasks because the particle data resides in the graphics memory and it will allow to render more particles because the communication limitation is no longer an issue.

There are two types of particle systems: stateless and state-preserving particle systems [32]. Stateless particle systems do not store the current positions and other data of the particles. New positions of particles are determined by a closed form function from the initial position and the current time. State-preserving particle systems allow using numerical, iterative integration methods to compute the particle data from previous values. Using this method it is also possible to have collision detection with dynamic objects in the environment. Our implementation is a state-preserving particle system.

6.2 Data Storage

The data of the particles can be split up into two major categories: positions and velocities. All this data is stored into two floating point textures. The RGB colour components of these textures will be treated like the x, y, and z coordinates. The textures are used as inputs to a shader to read the current positions and velocities, but they are also used as render targets to update the positions and velocities in every timestep. Because you can not read and write from the same texture in one rendering pass, double-buffering is used (cf. figure 6.1). This means you have a pair of each texture and the textures will be used in turn as input and output textures.

The user has the choice of creating 16bit or 32bit floating point textures. 32bit textures will increase the precision of the particles but will of course have an impact on the performance of the particle system.

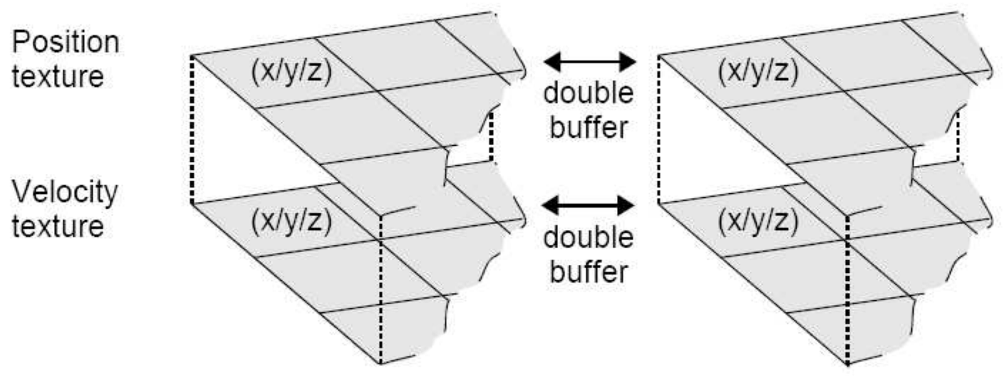


Figure 6.1: Particle storage in textures

6.3 Algorithm

6.3.1 Overview

The positions and velocities of the particles are stored in floating point textures. These textures will also be used as render targets. In a first step new particles will be created and old particles will be destroyed. Next, the velocities and positions of the particles will be updated. When the new positions are calculated they need to be copied from a texture to a vertex buffer so they can be transferred to the vertex shader where they will be rendered to the screen as point sprites or regular points. An overview is shown in figure 6.2.

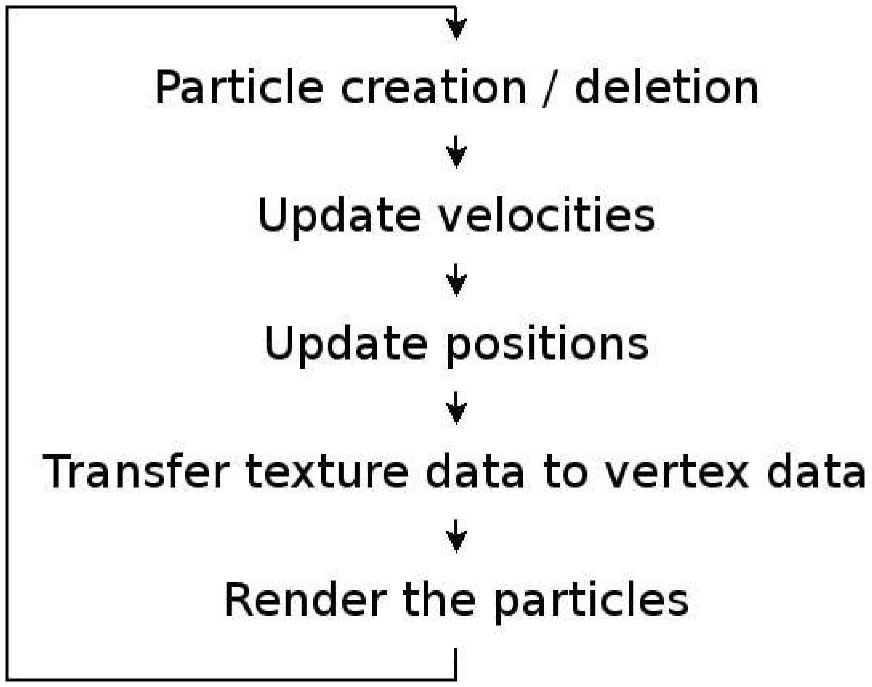

Particle creation / deletion
Update velocities
Update positions
Transfer texture data to vertex data
Render the particles

Figure 6.2: Algorithm overview

6.3.2 Step 1: Create and Destroy Particles

Original Method

The particles of a particle system do not have an eternal lifespan. Every timestep new particles are created which will replace the position in the textures of old particles. The data structure that holds the particles and their index in the textures is a queue. This queue will be stored in main memory so the available positions can be determined on the CPU. It is hard to do this on the GPU because allocation problems are serial by nature and can not be done efficiently with a parallel algorithm on the GPU.

Every timestep, a number of particles are added to the queue depending on the duration of the last frame. If the queue is already full, this means the maximum number of particles is reached because there are no more free indexes in the texture. We simply remove the first item in the queue because this is the oldest active particle. This method eliminates the need to manage the age of the particles because the oldest particles are automatically replaced by new ones.

The initial speed and velocities of the particles are determined by the behaviour of the particle system. The behaviour can be specified at the initialisation of the particle system. Finally the values of the new positions and velocites are written to the textures using a fragment program.

GPU Method

In an attempt to create an implemention of a particle system that runs completely on the GPU we tried to create an allocation mechanism that does map to the GPU.

The creation of new particles requires the use of random numbers to determine the initial velocity of the particles. Since random number generation is not supported on current GPUs, a preprocessing step on the CPU is needed to create an initial set of velocities based on the behaviour of the particle system. This set of velocities is stored in a floating point texture. To determine which particles need to be replaced a new field is added to the position texture storing the age of the particle. This field will be stored in the alpha value of the pixels.

The creation of new particles can now be done in a fragment shader. The arguments of the shader are the max age of the particles and three textures: one containing the pre-generated random velocities and two others containing the positions and the velocities. The shader will check the age of the current particle, if this value is bigger than the maximum age the particle will be replaced by a new one. This is done by performing a texture lookup in the random velocity texture to determine the new velocity of the particle and by resetting the position and age of the particle.

6.3.3 Step 2: Updating the Velocities

In the next step the velocities of the particles will be updated. This is done in a fragment shader that has two textures as arguments (positions and velocities). The shader will be executed for each pixel of the render target by rendering a screen-sized quad. The render target will be one of the two velocity textures (the one not used as input). Other parameters of the shader include the timestep, gravity, and the location and radius of the sphere.

First the shader performs two texture lookups to determine the current velocity and position of the particle. Next an estimate of the position in the following timestep will be made based on these values and with the help of an Euler integration. The Euler integration is shown in equation 6.1, where p_{est} is the new estimated position, p_{old} is the result from the texture lookup in the position texture, v_{old} is the result from the texture lookup in the velocity texture and Δt is the timestep.

$$p_{est} = p_{old} + v_{old} \cdot \Delta t \tag{6.1}$$

Next the global and local forces will be calculated and accumulated into a single vector so the acceleration can be calculated with Newtonian physics as shown in equation 6.2 with a the acceleration vector, F the accumulated force and m the mass of the particle. In our implementation, all particles have unit mass and the only global force is gravity. So gravity can be immediately used as the acceleration vector.

$$a = \frac{F}{m} \tag{6.2}$$

The next step is collision detection. Collision with complex objects is not very practical on GPUs. Collision with simple objects like a plane or a sphere however is pretty cheap to compute. Collision detection is done with the expected next position that was calculated before. The expected position is used instead of the current position to avoid particles getting caught inside a collider for one integration step. When collision is detected, the new velocity after the collision has to be computed. This is done by splitting up the current velocity into a normal and tangential component as shown in equations 6.3 and 6.4 where n is the normal of the collider at the collision point and v_{old} is the result from the texture lookup in the velocity texture.

$$v_n = (v_{old} \cdot n)n \tag{6.3}$$

$$v_t = v_{old} - v_n \tag{6.4}$$

When the new velocity is calculated two material properties are taken into consideration: dynamic friction μ and resilience ϵ. Dynamic friction influences the tangential component of the resulting velocity and simulates a reduction in velocity due to friction with the collider. Resilience influences the normal component of the new velocity and simulates the loss of returned energy caused by the collision. The new velocity resulting from the collision can now be computed with the formula of equation 6.5.

$$v_{new} = (1 - \mu)v_t - \epsilon v_n \tag{6.5}$$

A schematic representation of a collision with a sphere is shown in figure 6.3.

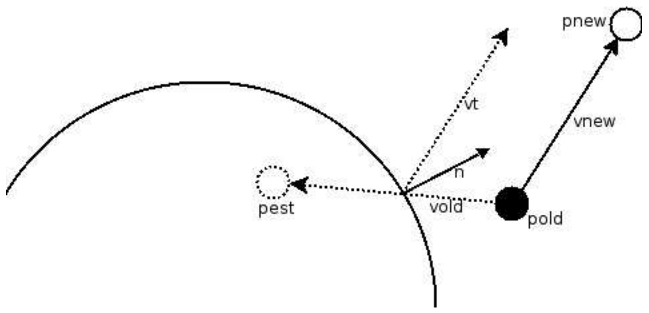


Figure 6.3: Particle collision with a sphere

6.3.4 Step 3: Updating the Positions

After we have determined the new velocities of the particles we can calculate the new positions. This is also done in a fragment shader, that has two textures as arguments (positions and velocities) and a floating point value that contains the timestep Δt. First the current position and velocity of the particles are looked up in the textures resulting in p_{old} and v_{new}. The new position can now be computed with an Euler integration as shown in equation 6.6.

$$p_{new} = p_{old} + v_{new} \cdot \Delta t \tag{6.6}$$

Note: When the hardware has support for multiple render targets (a fragment shader can output up to four values per rendering pass as specified in the ARB_draw_buffers extension [34]) the particle system will use this extension to update the positions and velocities of the particles at the same time. This means that steps 2 and 3 can be done in one rendering pass.

6.3.5 Step 4: Transfer Texture Data to Vertex Data

There are two possible ways to transfer texture data to a vertex shader: Vertex textures [35] or Vertex buffer objects [36].

Vertex textures require Vertex shader 3.0 and allow you to perform texture lookups from a vertex program just like you can do in a fragment program. This method seems rather optimal but because the lookup introduces a small latency [37] this method is only advisable when a fair amount of computation has to be performed on every lookup to hide the latency. Since that is not the case in our particle system it is actually faster to use Vertex Buffer Objects.

A vertex buffer object is a powerful extension that allows you to store certain data in high-performance memory on the server side [38]. This extension enables a very interesting optimization: Render to vertex array. This allows us to copy the position data stored in textures to a vertex buffer object which avoids copying the texture to the client side and putting it back on the server as input for a vertex shader as shown in figure 6.4.

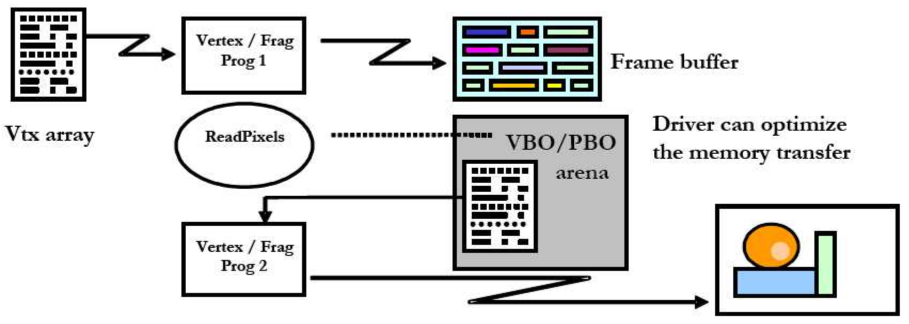


Figure 6.4: Render to vertex array

6.3.6 Step 5: Render the Particles

In a last step the particles are rendered to the screen in a vertex shader. The particles are rendered as point sprites using the ARB_point_sprite extension [39]. The extension allows you to render the particles as quads with the cost of only rendering a single point. The particles are rendered with the colour and size as is defined in the behaviour that is specified at the initialisation of the particle system. Images of the rendering result are shown in figure 6.5 and 6.6.

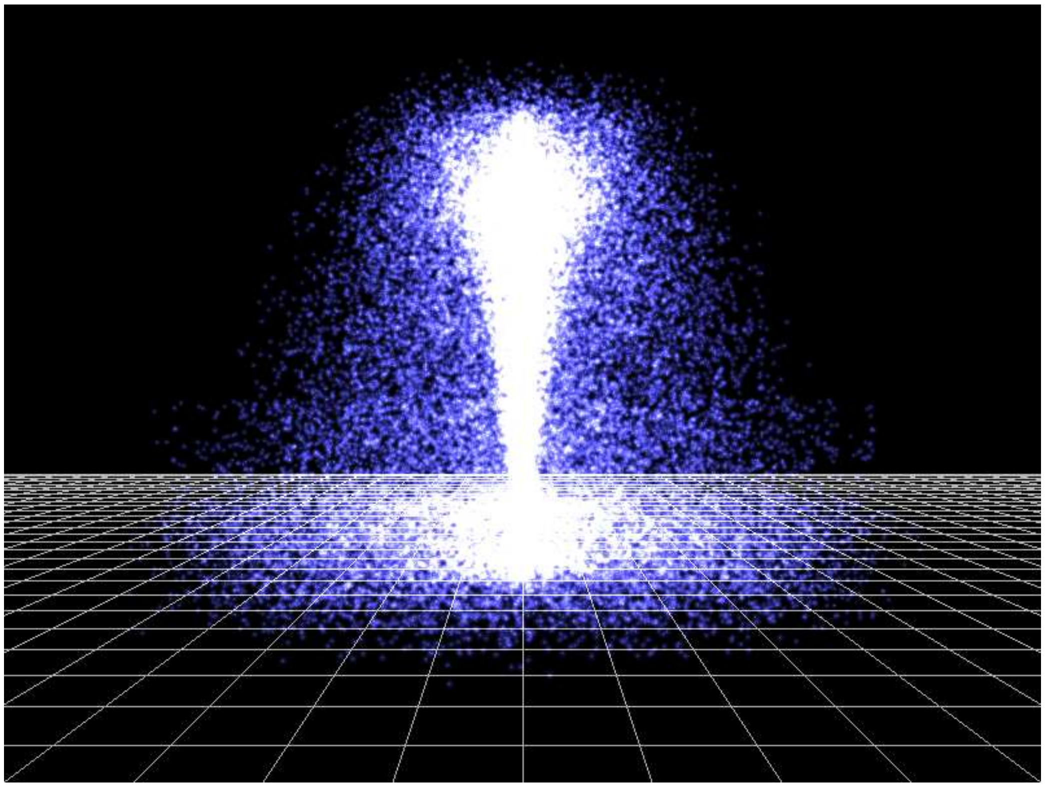


Figure 6.5: Particle system: fountain behaviour

6.4 Own Contributions

In this section we discuss some differences between our implementation and the already existing implementations of particle systems on the GPU.

6.4.1 Framebuffer Objects

Our implementation uses the EXT_FRAMEBUFFER_OBJECT extension [33] to render data directly to textures. This extension was approved by the ARB "superbuffers" working group on January 31, 2005 and has only been implemented in leaked beta drivers by NVIDIA ($\geq$ Forceware 75.95)[1]. Framebuffer objects (FBOs) are designed to be windowsystem- and vendor-independent. This allows them to be used on every platform that supports OpenGL in contrast to the WGL_ARB_render_texture [40] which is usually

[1]At the time of writing

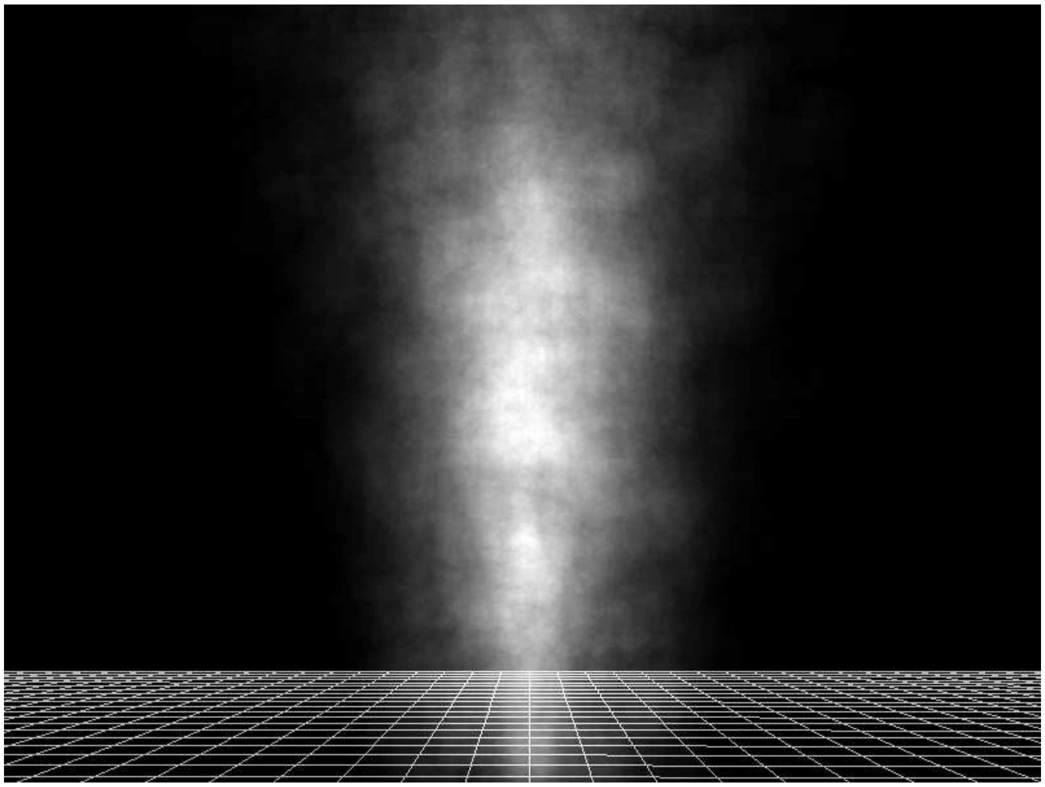


Figure 6.6: Particle system: smoke behaviour

used in combination with the WGL_ARB_pbuffer extension [41] to render directly to textures and is only supported on the Windows platform.

After creating a framebuffer object, textures can be attached as colour attachments to the framebuffer. Multiple textures can be added to the same FBO if they have the same format and size. This makes it possible to render to different textures in consecutive shaders whithout invoking a context switch. To set the target texture `glDrawBuffer()` needs to be called with the appropriate colour attachment as argument.

6.4.2 Particle Initialisation

As discussed earlier we also created a version of the particle system that performs the particle initialisation on the GPU. Existing implementations always use the CPU to manage the particles. On our implementation the CPU is only used to organise the graphics-API calls.

6.5 Results

In this section we show the resulting performance of the particle system we created by running some scenarios with a varying number of particles and precision. All the tests were run under Windows XP on an AthlonXP 2000 with 512MB ram equipped with an NVIDIA Geforce 6800 with 128MB ram (12 pixel pipelines, 5 vertex units) connected with an AGP 4x bus.

6.5.1 Original Method

The results of our implementation of the particle system using the original method are shown in table 6.1. The table shows some test scenarios with a varying numbers of particles and precision. A difference is also made between rendering the particles with simple GL_POINTS or rendering with point sprites.

The results are pretty straightforward, rendering more particles decreases the framerate because larger textures are needed which causes the shaders to perform more work. Using 16-bit precision is faster than using 32-bit precision. Except when a large number of point sprites are being rendered the precision does not seem to matter. The reason for this is that when a large number of point sprites need to be rendered, the application becomes limited by the fill rate of the graphics card because drawing lots of point sprites in a small region requires all these sprites to be blended together.

The results we obtained are satisfactory. We can simulate one million particles in realtime at 29 frames per second in 32-bit precision.

# particles	Precision	Frames per second	
		GL_POINTS	Point sprites
65536	16 bit	512	175
65536	32 bit	432	148
262144	16 bit	169	48
262144	32 bit	128	51
1048576	16 bit	46	15
1048576	32 bit	29	14

Table 6.1: Performance of the particle system using the original method

6.5.2 Full GPU Method

The results of the particle system using the GPU method to initialise the particles are shown in table 6.2. Again the table shows some test scenarios with a varying number of particles. This time only 32-bit is supported.

Again the results are pretty much what we expected, rendering more particles means less frames per second. If we compare this method with the original method we see that it is being outperformed by the original method, especially with fewer particles. But as the number of particles increases the GPU method starts to catch up with the original method. The reason that the original method is faster is because part of the work is done on the CPU. But as the number of particles gets above 1 million the difference gets smaller because then the CPU has to perform a fair amount of computation too.

# particles	Precision	Frames per second	
		GL_POINTS	Point sprites
65536	32 bit	365	194
262144	32 bit	115	53
1048576	32 bit	25	11

Table 6.2: Performance of the particle system using the GPU method for initialisation

Chapter 7

GPU-math Library

The implementation of the GPU-math library that we made is based on the paper of Jens Krüger and Rüdiger Westermann: Linear Algebra Operators for GPU Implementation of Numerical Algorithms [7].

7.1 Introduction

Numerical techniques for solving partial differential equations have a variety of applications in physics based simulation and modeling and are frequently used in computer graphics to provide realistic simulation of real-world phenomena. The downside is that because of the numerical complexity of these techniques they often exceed the limits of available memory and computational power. Therefore off-loading the computation to the GPU with its extreme power and parallelism should make it possible to perform real-time simulations with floating point precision.

A math library that performs matrix and vector operations on the GPU will aid in the implementation of these numerical techniques by providing an abstraction of the various operations, thus making it easier to convert numerical algorithms to the GPU. Available operations of the library are:

- matrix - matrix operations (addition, multiplication,...)
- matrix - vector multiplication

- vector reduction (min, max, sum,. . .)
- vector scale

7.2 Data Storage

7.2.1 Vectors

The easiest way to store vectors of length N would be to create 1xN 1D-textures, but this method is limited in several ways. The width and height of textures is limited by the hardware. By using 1D-textures the size of a vector would be limited by the maximum width of the texture. If you store the vector in a 2D-texture the size of the vector will be limited by the square of the maximum width, which allows you to create much bigger vectors. Secondly, square textures are rendered faster in graphics hardware than rectangular textures [7]. To allow even bigger vectors and to benefit from the SIMD-architecture of GPUs, RGBA textures will be used. (cf. figure 7.1)

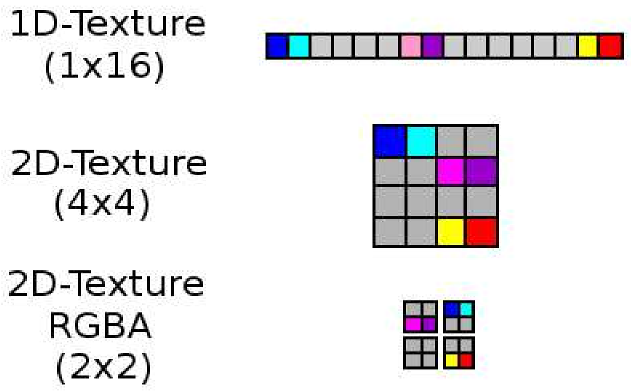


Figure 7.1: Vector representation

7.2.2 Dense Matrices

Dense matrices are matrices that have their non-zero values spread all over the matrix. Thus you can treat them as a set of column vectors. Each column will be stored in a vector which is a 2D-texture as indicated in the previous section. A schematic representation is shown in figure 7.2.

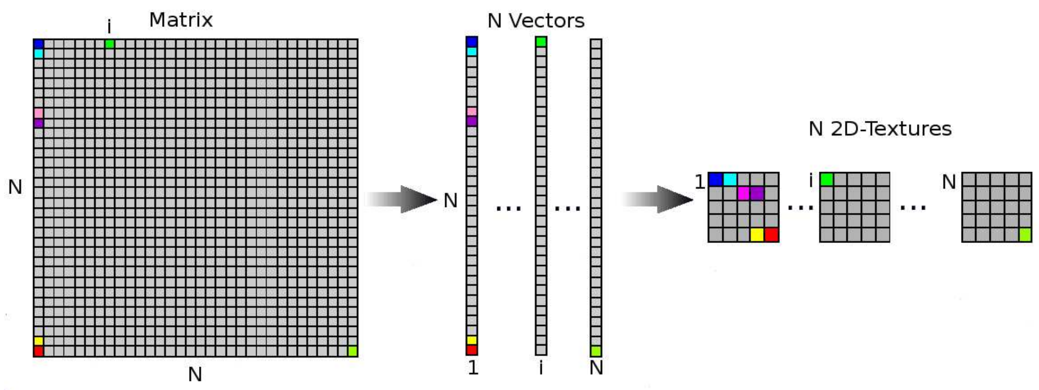


Figure 7.2: Dense matrix representation [7]

7.2.3 Banded Sparse Matrices

Banded sparse matrices are matrices that contain only zeros, accept for some bands around the diagonal. These matrices are treated as a set of diagonal vectors, so only the diagonals need to be stored in textures. A schematic representation is shown in figure 7.3. The bands to the left of the diagonal are stored from the beginning of the texture and zeros are appended to the end. Bands to the right of the diagonal need to be stored a bit differently, zeros need to prepended to the texture. The number of zeros depends on the position of the band. If it is the first band to the right of the diagonal 1 zero is prepended, if it is the second band to the right of the diagonal two zeros need to be prepended and so on. This shifting operation is done to ease the matrix - vector multiplication that is discussed in the next section.

7.3 Operations

7.3.1 Vector Operations

Basic artithmetic operations on two vectors can be performed in simple fragment shaders. Both vectors are passed as arguments to the shader and their

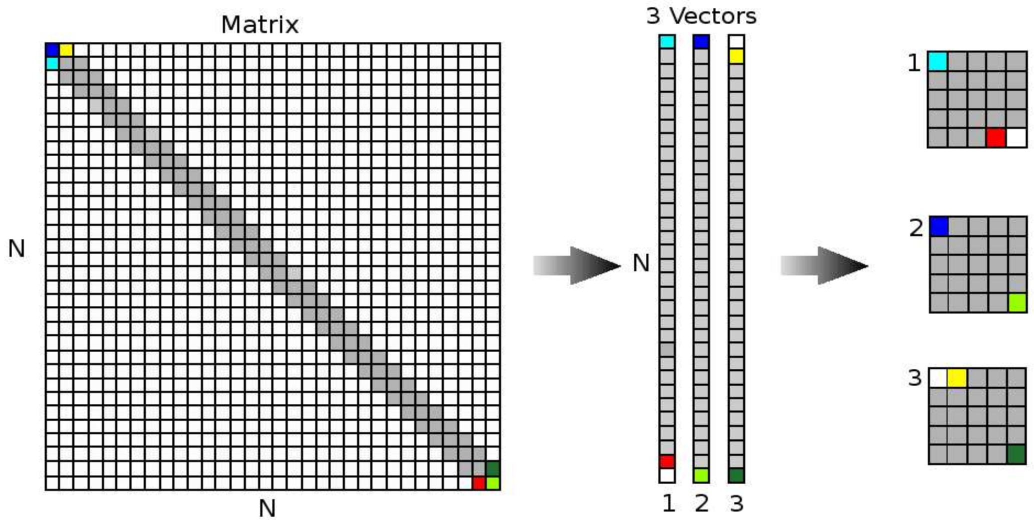


Figure 7.3: Banded sparse matrix representation

values will be combined. Supported operations are addition, subtraction and multiplication. Each operation is defined in different shader. This approach is used in favor of passing the operation as an argument to avoid costly branching operations. The product of a vector and a scalar is done passing the scalar as an argument to the fragment shader. This results in the following list of available functions:

```
clVecRGBA & add(clVecRGBA & v1, clVecRGBA & v2, float a = 1, float b = 1)
clVecRGBA & sub(clVecRGBA & v1, clVecRGBA & v2, float a = 1, float b = 1)
clVecRGBA & mult(clVecRGBA & v1, clVecRGBA & v2, float a = 1, float b = 1)
clVecRGBA & scale(clVecRGBA & vec, float scalar)
```

the two float values a and b that are passed as arguments to the *add*, *sub* and *mult* functions are multiplied with $v1$ and $v2$ respectively before the computation.

7.3.2 Matrix - Vector Multiplication

Dense Matrices

Matrix vector multiplication is done by multiplying every i-th column of the matrix with the i-th element of the vector and then adding these results together. An example is shown in figure 7.4. The multiplication is performed in a fragment shader that performs several columns per pass by using multi-texturing. Several columns (2D-textures) of the matrix are passed as arguments to the shader. The vector that will be used in the multiplication is also passed as a parameter. The texture coordinates that need to be used to lookup the i-th element of the vector, used to multiply with the i-th column of the vector, are calculated on the CPU and are passed to the shader as uniform paramaters. In order to be able to save the output of the shader and use the intermediate result in the next pass as input to the shader double-buffering will be used.

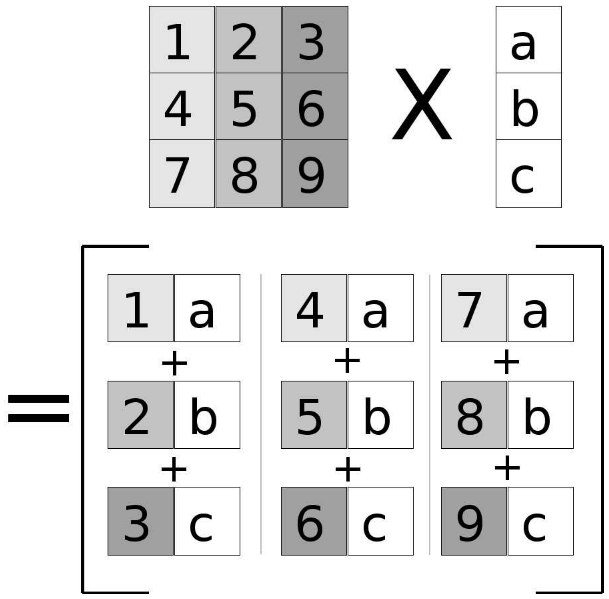


Figure 7.4: Dense matrix - Vector multiplication

This results in the following list of available functions:

```
clVecRGBA & matVecOp(clDenseMatRGBA & A, clVecRGBA & x)
clVecRGBA & matVecOp(clMath::Op op, clDenseMatRGBA & A, clVecRGBA & x, clVecRGBA & y)
```

The first function simply performs the computation of dense matrix A with vector x. The second function performs the computation of Ax *op* y where *op* can be CL_ADD (addition), CL_SUB (subtraction) or CL_MULT (multiplication).

Banded Sparse Matrices

Banded sparse matrices can be multiplied with a vector in one pass. All the bands and the vector are passed as arguments to the shader. A representation is shown in figure 7.5. Every pixel from the bands is multiplied with the corresponding pixel of the vector. This is possible because we already shifted the bands that appear on the right side of the diagonal to ensure the red, green, blue and alpha channels of the pixels can be directly multiplied with each other. Next we need to add the right values from the result of the multiplication together to get the final result. The values that need to be added together are presented in the same colour in figure 7.5. The values inside the square are the results of the calculations from the current pixel. However, some values needed to achieve the resulting pixel are outside of this square. This is solved by also performing a texture lookup for the previous and the next pixels of the bands and the vector and perfoming the same multiplication as we did for the current pixel. Finally the values of the same colour can be added together to get the value of the output pixel.

Because the values that need to be added together depend on the number of bands different shaders have been written for different numbers of bands. This also means that the maximum number of bands 15 because on current hardware, multi-texturing is only supported for up to 16 textures and one texture is used for the vector, so 15 textures remain for the bands.

This results in the following list of available functions:

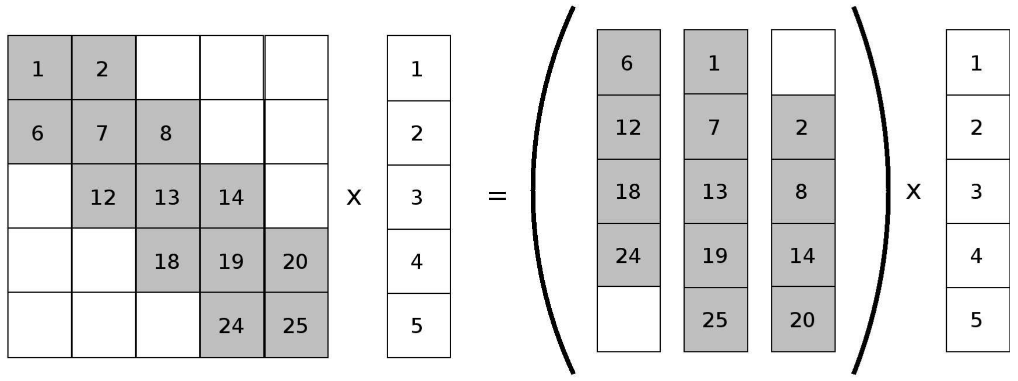
```
clVecRGBA & matVecOp(clBandedSparseMatRGBA & A, clVecRGBA & x)
clVecRGBA & matVecOp(clMath::Op op, clBandedSparseMatRGBA & A, clVecRGBA & x, clVecRGBA & y)
```

The first function simply performs the computation of banded sparse matrix A with vector x. The second function performs the computation of $Ax\ op\ y$ where *op* can be CL_ADD (addition), CL_SUB (subtraction) or CL_MULT (multiplication).

7.3.3 Matrix - Matrix Multiplication

Dense matrices

Matrix - matrix multiplications can be completely computed with the functionality that is already present in the library. Suppose we have two matrices



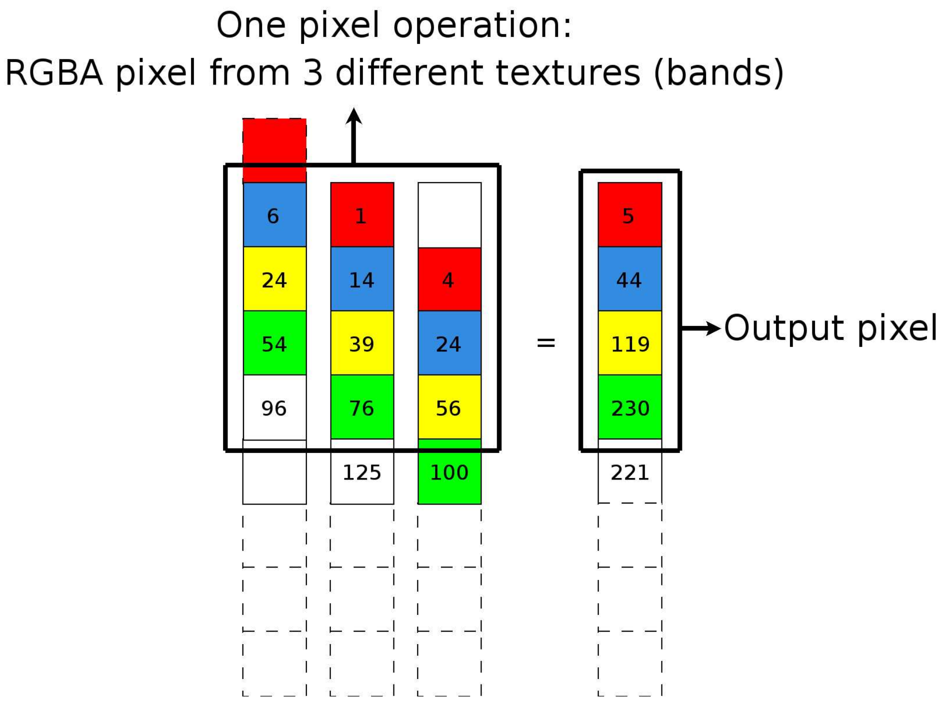


Figure 7.5: Banded sparse matrix - Vector multiplication

A and B. Then the resulting matrix C will be computed by multiplying A with every row of B by calling the matVecop method and adding the resulting vectors as rows to C.

This results in the following function:

```
clDenseMatRGBA & matMatOp(clDenseMatRGBA * m1, clDenseMatRGBA * m2);
```

The function simply takes two dense matrices as arguments and computes the product.

7.3.4 Vector Reduction

Sometimes it is necessary to combine all the values of a vector into a single value, e.g. computing the maximum, minimum, sum, . . . of all the values. Therefore a reduce operation is added to the library.

The algorithm starts with a 2D-texture that contains the values of the vector. Two textures are created that will act as a double-buffer. In each step a quadrilateral is rendered that is half the size of the previous quadrilateral. The values are written to the current drawbuffer. Each pixel of the rendered texture will contain a combination of four pixels of the previous texture. This means that in every pass four texture lookups are performed. An operation will be applied to these values and the result will be written to the render target. This is repeated untill the size of the quadrilateral is 1x1, the final pixel then contains the result of the reduction. The technique is shown in figure 7.6. Available operations are CL_ADD, CL_MULT, CL_MAX, CL_MIN, CL_NORM. For a vector that is represented in a texture of dimension 2^n, n rendering passes need to be performed to compute the resulting value.

This results in the following list of available functions:

```
float reduceVecOp(clMath::Op op, clVecRGBA & v1);
float reduceVecOp(clMath::Op op, clVecRGBA & v1, clVecRGBA & v2);
```

The first function will perform the reduce operation on vector $v1$ with operation *op*. The second function will perform the reduce operation on the product of $v1$ and $v2$ with operation *op*.

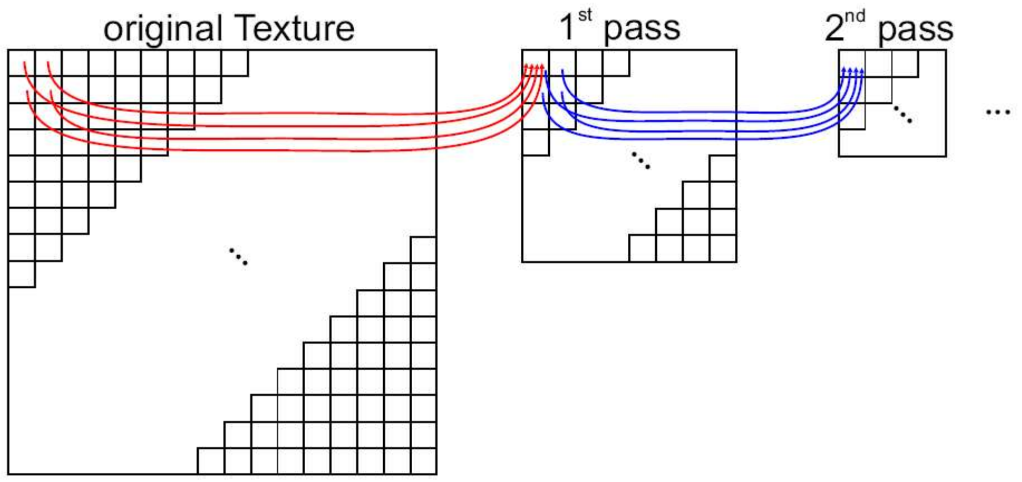


Figure 7.6: Vector reduction [7]

7.4 Own Contributions

7.4.1 Framebuffer Object

We made use of the EXT_FRAMEBUFFER_OBJECT extension for rendering to off-screen buffers. This extension was already explained in section 6.4.1

7.4.2 Adjusted Vector Reduction

In the original paper the vector reduction was performed by rendering every intermediate result into a new texture that was half the size of the currently active texture. Instead we used a double-buffering approach with fixed size textures, but simply halved the viewport on every pass. This resulted in a performance gain because less textures had to be created and less context switches were needed.

7.4.3 Adjusted Dense Matrix - Vector Multiplication

Instead of using the diagonal approach to represent dense matrices, we have chosen to use a column based approach. This made it easier to perform dense matrix - vector multiplications when using rgba-textures to store the data.

7.4.4 New Banded Sparse Matrix Method

We created a new method for the banded sparse matrix - vector multiplication. It proved to be much faster than the dense matrix approach as is shown in section 7.5.5

7.5 Results

In this section we will show the results of the benchmarks that were performed with the math-library. All the tests were run under Linux on an AthlonXP 2000 with 512MB ram equipped with an NVIDIA Geforce 6800 with 128MB ram (12 pixel pipelines, 5 vertex units) connected with an AGP 4x bus. For comparison with CPU performance we used Matlab.

7.5.1 Vector - Vector Operations

To test vector-vector operations we simply took two vectors, multiplied both vectors with a scalar value and then added them together a number of times. Results are shown in table 7.1.

The CPU outperforms the GPU implementation when small vectors are used, this is because the overhead of creating textures and framebuffer objects on the GPU is too big for the little amount of computation that needs to be performed. When the size of the vectors increases, more computation needs to be performed, eliminating the overhead of the GPU implementation. For vectors of 1000000 elements the GPU implementation runs approximately 20 times faster than the CPU.

7.5.2 Vector Scale Operation

To test the vector scale operation a vector was multiplied with a scalar value a number of times. Results are shown in table 7.2.

Vectorsize	Iterations	Time (seconds)		Factor
		GPU	CPU	
100	50000	2.970	0.134	0.05
10000	50000	2.960	5.356	1.81
50000	50000	4.620	92.125	19.84
100000	50000	9.120	185.046	20.29
500000	50000	44.530	937.030	21.04
1000000	50000	86.390	1853.600	21.46

Table 7.1: Timing of vector - vector operations

The vector scale operation shows similar results to the vector - vector operations. The GPU is slower for small vectors due to the overhead but faster for larger vectors. For a vector of 1000000 elements the GPU implementation is 25 times faster than the CPU. The vector scale operation performs slightly better than the vector - vector operation because only one texture lookup has to be performed per kernel execution instead of two.

Vectorsize	Iterations	Time (seconds)		Factor
		GPU	CPU	
100	50000	1.279	0.031	0.02
10000	50000	1.279	1.610	1.26
50000	50000	2.844	57.031	20.05
100000	50000	5.510	118.391	21.45
500000	50000	24.490	544.220	22.22
1000000	50000	50.840	1306.090	25.69

Table 7.2: Timing of vector scale operation

7.5.3 Vector Reduce Operation

To test the vector reduce operation a vector was reduced a number of times. Results are shown in table 7.3.

Again we see that the CPU version is much faster when small vectors are used but it becomes slower when the vector size increases. Reducing a vector of 1000000 elements is 7 times faster on the GPU than on the CPU. The speedup is smaller than in previous operations because a reduce operation requires multiple render passes. And since double-buffering is used the render target needs to be changed each pass which also involves a cost. What might seem weird is that reducing a vector of 500000 elements takes just as long as reducing a vector of 1000000 elements. This is actually normal since both vectors are stored in textures that have quadratic dimensions, so both vectors are stored in 512x512 textures that are completely updated.

Vectorsize	Iterations	Time (seconds)		Factor
		GPU	CPU	
100	10000	2.170	0.027	0.01
10000	10000	3.470	0.879	0.25
50000	10000	3.900	7.047	1.81
100000	10000	6.530	15.144	2.74
500000	10000	19.740	74.346	3.77
1000000	10000	19.750	148.297	7.51

Table 7.3: Timing of vector reduce operation

7.5.4 Dense Matrix - Vector Multiplication

We performed multiplications of dense matrices with vectors. The results are shown in table 7.4

The dense matrix - vector multiplication is always slower on the GPU than on the CPU. The main reason for the bad performance is the number of rendereing passes that need to be executed. Dense matrices require a lot of rendering passes because only 12 columns can be multiplied per pass.

Matrixsize	Iterations	Time (seconds)		Factor
		GPU	CPU	
100x100	10000	5.797	0.250	0.04
200x200	10000	11.343	3.078	0.27
400x400	10000	25.422	14.500	0.57
1000x1000	10000	107.880	99.954	0.93

Table 7.4: Timing of dense matrix - vector multiplication

7.5.5 Banded Sparse Matrix - Vector Multiplication

The same tests as in the previous section were performed, instead now we used banded sparse matrices. The results are shown in table 7.5

Since banded sparse matrices can be multiplied with vectors in one pass the performance is better than the dense matrix multiplication. The more bands the matrix contains, the better the GPU performance will be because more computation needs to be performed per pass which allows the GPU to better hide the overhead.

Size	Iter.	Time (seconds)					
		1 band			3 bands		
		GPU	CPU	Factor	GPU	CPU	Factor
100x100	100000	3.875	0.937	0.24	4.719	1.500	0.32
200x200	100000	3.859	1.437	0.37	3.859	2.547	0.66
400x400	100000	4.265	2.234	0.52	5.125	4.515	0.88
1000x1000	100000	6.500	4.828	0.74	7.830	10.703	1.37
		5 bands			7 bands		
100x100	100000	5.235	2.047	0.39	5.625	2.570	0.46
200x200	100000	5.188	3.640	0.70	5.625	4.765	0.85
400x400	100000	5.578	6.890	1.24	6.125	9.016	1.47
1000x1000	100000	8.660	16.375	1.89	9.630	22.297	2.32

Table 7.5: Timing of banded sparse matrix - vector multiplication

7.5.6 Conjugate Gradient Method

Using the available operations of our library, we created a Conjugate Gradient (CG) solver on the GPU. The CG method is the most popular iterative method for solving large systems of linear equations. CG is effective for systems of the form $Ax = b$ where x is an unknown vector, b is a known vector, and A is a known, square, symmetric, positive-definite (or positive-indefinite) matrix [42].

The results of the method when using dense matrices are shown in table 7.6. Again we can see that for small matrices the CPU is faster than the GPU. However, when we increase the size of the matrix the GPU starts to perform better than the CPU. At sizes of 1800x1800, the GPU is twice as fast as the CPU. The main loop of the algorithm consists of four reduce operations, one matrix - vector multiplication and 2 vector - vector operations. The biggest slowdown factor is the matrix - vector operation since it is by far the least performant operation of the three, even if it is only executed once. The cg-method starts to catch up to the CPU at sizes of 1000x1000 because at those sizes the dense matrix vector -multiplication starts to catch up to the CPU too, as shown in section 7.5.4

Matrixsize	Iterations	Time (seconds)		Factor
		GPU	CPU	
100x100	10	10.734	0.125	0.01
500x500	10	16.531	6.031	0.36
1000x1000	10	22.707	20.625	0.91
1800x1800	10	32.640	65.671	2.01

Table 7.6: Timing of the conjugate gradient method with dense matrices

Chapter 8

Conclusion

In this thesis we studied the use of graphics hardware for general-purpose computation. We learned that modern GPUs contain very powerful processors that can be used for more than just graphics.

We gave some examples of research areas in which the GPU has been successfully used and we created two applications that use the GPU for general-purpose tasks. One is a particle system of which the particles are updated on the GPU. The system can render over a million particles in real-time. The other was a math-library that performs matrix and vector operations on the GPU. In general we saw a performance increase when comparing the results to the CPU. Some operations where even 25 times faster than their CPU equivalent.

Our results confirmed that the GPU can indeed be used for general-purpose tasks, but before the GPU can be widely used as a general purpose processor some barriers need to be broken:

- Better debuggers and profilers need to be developed
- A GPU programming language is needed that provides an abstraction to the graphics API
- Platform independence between different graphics vendors (our implementations do not run on ATI hardware)

Our guess is that GPUs will mostly be used to perform general-purpose tasks for computations that need to be transferred to the GPU anyway. Our particle system is such an example.

Bibliography

[1] GPGPU website. http://www.gpgpu.org/.

[2] Zhaowei Fan, Huagan Wan, and Shuming Gao. Streaming collision detection using programmable graphics hardware, 2003.

[3] Cyril Zeller. Introduction to the hardware graphics pipeline. Presentation at Eurographics 2004.

[4] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[5] Randy Fernando. Trends in GPU evolution. Presentation at Eurographics 2004.

[6] Mark Harris. GPGPU: General purpose computation on GPUs. Presentation at the Game Developers Conference 2005.

[7] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *Proceedings of ACM SIGGRAPH 2003*, volume 22(3) of *ACM Transactions on Graphics*, pages 908–916, 2003.

[8] Randima Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional, March 2004.

[9] João Luiz Dihl Comba, Carlos A. Dietrich, Christian Augusto Pagot, and Carlos Eduardo Scheidegger. Computation on gpus: From a programmable pipeline to an efficient stream processor. *RITA*, 10(1), 2003.

[10] Matt Pharr and Randima Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, March 2005.

[11] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[12] *Cg Toolkit, User's manual: A developer's guide to programmable graphics*, January 2004.

[13] John Kessenichand Dave Baldwin and Randi Rost. *The OpenGL Shading Language*, April 2004.

[14] K. Hillesland and A. Lastra. GPU floating-point paranoia. In *GP^2*, 2004.

[15] Randy Fernando. GPGPU: General-purpose computation using graphics hardware. Presentation at I3D 2005.

[16] Dx.next: The near and nearest future of hardware graphic acceleration. http://www.digit-life.com/articles2/dx-next/.

[17] Directx next early preview. http://bink.nu/Article606.bink.

[18] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[19] Aaron Lefohn, Ian Buck, John Owens, and Robert Strzodka. GPGPU: General purpose computation on graphics processors. Presentation at IEEE Visualization 2004.

[20] Brandon Lloyd, Dinesh Manocha, Ming Lin, Naga K. Govindaraju, and Wei Wang. Fast computation of database operations using graphics, March 21 2004.

[21] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware acceleration for spatial selections and joins. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 455–466, New York, NY, USA, 2003. ACM Press.

[22] Marcin Jędrzejewski. Computation of room acoustics on programmable video hardware, 2004.

[23] BionicFX announces audio processing on NVIDIA gpu, 2004.

[24] Timothy John Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, March 2004.

[25] Karl E. Hillesland, Sergey Molinov, and Radek Grzeszczuk. Nonlinear optimization framework for image-based modelling on programmable graphics hardware. *ACM Trans. Graph.*, 22(3):925–934, 2003.

[26] Nigel Stewart, Geoff Leach, and Sabu John. Improved CSG rendering using overlap graph subtraction sequences. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 47–53, New York, NY, USA, 2003. ACM Press.

[27] Valerio Pascucci. Isosurface computation made simple. In *VisSym*, pages 293–300, 2004.

[28] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 112–119. Eurographics Association, 2003.

[29] Jens Krueger and Ruediger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[30] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *SC'2004 Conference CD*, Pittsburgh, PA, November 2004. IEEE/ACM SIGARCH. Stony Brook University.

[31] Youquan Liu, Xuehui Liu, and Enhua Wu. Real-time 3d fluid simulation on the gpu with complex obstacles, 2004.

[32] Lutz Latta. Building a million particle system, 2004.

[33] *EXT_framebuffer_object*.
http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt .

[34] *ARB_draw_buffers specification*.
http://oss.sgi.com/projects/ogl-sample/registry/ARB/draw_buffers.txt .

[35] *ARB_vertex_shader specification*.
http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_shader.txt .

[36] *ARB_vertex_buffer_object specification.*
http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt .

[37] Philipp Gerasimov, Randima (Randy) Fernando, and Simon Green. Shader model 3.0: Using vertex textures, June 2004.

[38] Using vertex buffer objects (vbos), October 2003.

[39] *ARB_point_sprite specification.*
http://oss.sgi.com/projects/ogl-sample/registry/ARB/point_sprite.txt .

[40] *WGL_ARB_render_texture.*
http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_render_texture.txt .

[41] *WGL_ARB_pbuffer.*
http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_pbuffer.txt .

[42] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

Appendix A

Dutch Abstract

De Grafische Verwerkings eenheid (GPU)

Het verschil tussen een GPU en een grafische versneller zit hem in het feit dat GPUs niet enkel grafische bewerkingen versnellen, maar dat ze bepaalde fases uit de grafische pijplijn programmeerbaar maken.

GPUs zijn de laatste jaren zeer snel geworden. De laatste generatie GPUs kunnen viermaal zoveel floating point operaties uitvoeren per seconde dan de laatste generatie CPUs. Bovendien kan de GPU inkomende data in parallel verwerken, omdat hij uitgerust is met meerdere vertex- en pixelprocessors.

Door de komst van de high-level programmeertalen voor GPUs is het bovendien veel intuïtiever geworden voor de programmeur om shaders te maken. Hierdoor kunnen shaders sneller aangemaakt worden en wordt het gemakkelijker om de code te herbruiken en te debuggen. Voeg hier aan toe dat de laatste generatie videokaarten 32-bit floating point operaties ondersteunen over gans de grafische pijplijn en je bekomt een interessant apparaat om preciese bewerkingen op uit te voeren.

Er zijn ook nadelen verbonden aan het werken op de GPU. Zo is het noodzakelijk om gebruik te maken van een grafische API, je hebt een beperkte geheugeninterface en er is nog een gebrek aan degelijke debuggers en profilers. Bovendien is de bandbreedte om data van de GPU terug naar de CPU te halen beperkt.

Programmeren op de GPU

Programmeren op de GPU gebeurt volgens het stream programming model. Dit houdt in dat alle invoerdata gelezen worden als een stroom van elementen en elk element van de stroom is een data record waarop dezelfde bewerking dient te worden uitgevoerd. Het systeem voert dan een programma uit voor ieder element in de invoerstroom en plaatst het resultaat in een uitvoerstroom.

Een aantal beschikbare operaties op de GPU zijn: (random access) read-only geheugen, per-data-element interpolanten, tijdelijke registers, read-only constanten, write-only geheugen en floating point mathematische operatoren.

Operaties die momenteel niet beschikbaar zijn: stack, heap, operaties op integers, scatter operatie, reducties en een beperkt aantal outputwaardes per pixel. Sommige van deze operaties zijn te omzeilen, maar dit brengt een kost met zich mee.

Onderzoeksgebieden

GPUs werden al met succes ingezet in verschillende onderzoeksgebieden zoals daar zijn databases (uitvoeren van queries), audio- en signaalverwerking (voortplanting van geluid berekenen), geavanceerde rendering (ray tracing), computationele geometrie (collision detection), beeld- en volumeverwerking (fast fourier transform) en wetenschappelijk onderzoek (vloeistofsimulatie)

Voorbeeld: Collision detection

In hoofdstuk 5 wordt een gedetailleerde bespreking gegeven van een paper die collision detection algoritme op de GPU beschrijft. Het algoritme voert collision detection uit op sterlichamen.

Het algoritme bestaat uit drie stappen. In een eerste stap worden stralen gegenereerd vanuit het centrum van een lichaam. Daarna wordt er gecontroleerd of deze stralen intersecteren met een driehoek van een ander lichaam. Vervolgens gebeurt er een vergelijking van de threshold waardes.

Verder worden er nog een aantal optimalisatietechnieken besproken die de performantie van het algoritme verhogen (direction cone, backface culling, viewing volume culling, triangle caching and organization).

Particle Systeem

In hoofdstuk 6 bespreken we onze implementatie van een Particle Systeem op de GPU. De data van de particles, zoals positie en snelheid, wordt opgeslagen in textures waarvan de rgb-waardes gebruikt worden om de xyz-coordinaten in op te slaan.

Het algoritme van het particle systeem bestaat uit 5 stappen. In de eerste stap worden nieuwe particles aangemaakt en oude verwijderd. Daarna worden de snelheden van de particles en vervolgens worden de posities aangepast. In de vierde stap wordt de texture data die de posities bevat omgezet naar vertex data. In de laatste stap worden de particles dan gerendered aan de hand van deze vertex data.

Op deze manier kunnen ruim een miljoen particles in realtime gerendered worden op de laatste generatie grafische kaarten.

GPU-math bibliotheek

In hoofdstuk 7 bespreken we onze implementatie van een math-bibliotheek op de GPU. Deze bibliotheek ondersteunt allerlei matrix en vector operaties die gebruikt kunnen worden voor het oplossen van numerieke methodes op de GPU.

Beschikbare operaties van de bibliotheek zijn:

- matrix - matrix operaties (optellen, vermenigvuldigen,...)
- matrix - vector vermenigvuldiging
- vector reductie (min, max, som, norm,...)
- vector scalering

Uit de resultaten blijkt dat het voor grote vectors en matrices zeker de moeite kan zijn om deze bewerkingen op de GPU uit te voeren. In sommige gevallen was de GPU-implementatie zelfs tot 25 maal sneller dan de CPU-versie. Voor kleine data blijkt de gegenereerde overhead te groot te zijn om snelheidswinst te boeken.