The GPU as a high performance computational resource

Tor Dokken
SINTEF ICT, Applied Mathematics
P.O. Box 124 Blindern
0314 Oslo, Norway
Phone: +47 22 06 73 00
tor.dokken@sintef.no

Trond R. Hagen
SINTEF ICT, Applied Mathematics
P.O. Box 124 Blindern
0314 Oslo, Norway
Phone: +47 22 06 73 00
Trond.R.Hagen@sintef.no

Jon M. Hjelmervik
SINTEF ICT, Applied Mathematics
P.O. Box 124 Blindern
0314 Oslo, Norway
Phone: +47 22 06 73 00
Jon.A.Mikkelsen@sintef.no

ABSTRACT

With the introduction in 2003 of standard GPUs with 32 bit floating point numbers and programmable Vertex and Fragment processors, the processing power of the GPU was made available to non-graphics applications. As the GPU is aimed at computer graphics, the concepts in GPU-programming are based on computer graphics terminology, and the strategies for programming have to be based on the architecture of the graphics pipeline. At SINTEF in Norway a 4-year strategic institute project (2004-2007) *"Graphics hardware as a high-end computational resource"*, http://www.math.sintef.no/gpu/ aims at making GPUs available as a computational resource both to academia and industry. This paper addresses the challenges of GPU-programming and results of the project's first year.

Categories and Subject Descriptors

G.4 MATHEMATICAL SOFTWARE, *Parallel and vector implementations.*

General Terms

Algorithms, Performance, Experimentation.

Keywords

GPU, Geometry, Partial Differential Equations, Linear Algebra.

1. INTRODUCTION

Since the early days of computers, computer graphics has been essential for presenting results from classes of computer programs. While some applications only need a 3D graphics pipeline using CPU resources, other applications demand a hardware accelerated 3D graphics pipeline. During recent years advanced 3D computer graphics has become an integral part of many computer games, resulting in a huge market for affordable 3D graphics cards. Currently programmable 3D graphics cards for PC are priced between 200€ and 500€.

Within SINTEF ICT, Department of Applied Mathematics, 3D graphics has been an important support tool both for the activities within Computer Aided Geometric Design (CAGD) and Partial Differential Equations (PDEs). When GPUs with 32bit floating point arithmetic and programmable vertex and fragments shaders were introduced in 2003, we realized that these GPUs could be an important computational resource both within CAGD and PDE based simulation. Consequently we applied for support from the Norwegian Research Council to investigate these possibilities. Our application was successful, and we were awarded a Strategic Research Project "*Graphics hardware as a high-end computational resource*", http://www.math.sintef.no/gpu/, for the period 2004-2007.

The project is now into its second year, and our knowledge on GPU programming is growing. As the origin of the GPU is from computer graphics the terminology related to GPU programming has a strong computer graphics flavor. One of the ambitions of the project is to open GPU-programming to those outside of computer graphics. We hope that this paper is a step in this direction.

We will tell more about the Norwegian GPU-project in Section 2, and in Section 3 we address some of the project results. Then in Section 4 we will look at the graphics pipeline and a give short introduction to concepts and languages of GPU-programming. Section 5 will look into fragment shaders and how these can be used for executing "loop" structures. Some examples of GPU-applications will be given in Section 6.

2. The Norwegian GPU-project

The total budget for the project is approximately 1.5 M€, with 63% allocated for work by SINTEF employed scientists, and 37% allocated to two Ph.D. fellowships and one post doc. Fellowship. The project is not a university type project, but a project combining the industrial R&D institute SINTEF http://www.sintef.no/ , and fellowship with an industrial focus.

The project focuses on algorithms for applications of GPUs within

- Image processing
- Partial differential equations
- Geometry
- Linear algebra

The project uses standard PCs with commercially available graphics card. Until the middle of 2005 only PCs with AGP bus have been used. However, we expect soon to start using PCs with PCI-Express bus.

The project is coordinated by SINTEF ICT, Department of Applied Mathematics. The other active partners in the project are: The Center of Mathematics for Applications at the University of Oslo, http://www.cma.uio.no/; Department of Mathematics, the

University of Bergen, http://www.mi.uib.no/; and Narvik University College, http://www.hin.no/.

3. Some project results

Most visible results so far have been in the solution of partial differential equations using explicit finite difference and finite volume schemes. For solving partial differential equations we have observed a speedup by a factor between 10 and 20 compared to a Pentium 2.8 GHz CPU. For small computational grids the overhead of initializing shaders reduce the speedup. The speedup is best for shaders with many floating point instructions, while for schemes with few such instructions prefetching of data is not fast enough to ensure maximal use of the computational power. The best speedup we have observed is for 2. and 3. order high-resolution schemes solving the Euler equations that model the dynamics of compressible gasses [2]. These shaders are considerably more complex than shaders for standard 2. and 3. order schemes.

We have started to look into how the GPU can be used as an accelerator for CAD-type intersection algorithms, and filed a patent related to this idea. For such algorithms the fragment processor can be used for subdividing surfaces to create both a tight hull containing the surface, and a tight hull containing the normal field of the surface.

- The depth buffer of the GPU is a tool for testing for overlap of the extent of the hulls containing the surfaces. If no overlap, no intersection is possible.
- For two surfaces to intersect in a closed loop, the normal fields have to overlap. Thus using the depth buffer to determine that the hulls containing the normal field of surfaces do not intersect will rule out the possibility of closed intersection curves.
- As the shaders can also return information of the regions of possible overlap, important information on the actual intersection can be produced by the GPU.

The integration of shaders into intersection algorithms, and the use of shaders for intersection algorithm acceleration, are considerably more complex than the examples shaders for partial differential equations in Sections 6.1 and 6.2.

4. Programmable part of graphics pipeline

The programmable parts of recent GPUs are:

- The vertex processor, executing programs denoted vertex shaders.
- The fragment processor, executing programs denoted fragment shaders.

The basic data types of both the vertex and the fragment shaders are vectors of length 4 and 4x4 matrices. Standard operations on these are efficiently implemented in hardware. Depending on the actual GPU many other functions can also be hardware accelerated, e.g., elementwise logarithmic and trigonometric functions. A major advantage of such functions is that they allow the GPU to do the operations in parallel since the result in one vector element is independent of the other elements.

4.1 The fragment processor

Most of the programmable computational power of GPUs is available in the fragment processor. As an example, the nVidia GeForce 6 Series GPUs, introduced in 2004, have up to 16 pipelines in the fragment processor. Each pipeline can handle 4 floating point operations in parallel running at a maximum of 450MHz. Synthetic tests (programs made only for demonstrating computational power) show a total computational power of up to 50GFlops for the GeForce 6 Series GPUs. It is expected that performance will continue to grow by increasing the number of pipelines as well as the clock frequency, both in GPUs from nVidia and ATI.

Newer generations of GPUs support dynamic branching both in the vertex and the fragment processors. The fragment pipelines are handled synchronously in the GPU, and each of the fragment pipelines execute the same path. In an “if/else structure” we risk that both branches are executed, thus not saving execution time. The handling of “if/else structures” depends on the actual GPU and is expected to change with new generation of GPUs. The potential bottlenecks introduced by dynamic branching can be avoided as follows:

- In many cases it is possible to rewrite the shader to use arithmetic expressions instead of dynamic branching. This is especially advantageous when combined with the vector capabilities of the GPU.
- Early exit. If parts of the shader can be omitted for large parts of the primitive being rendered, dynamic branching can increase the performance. An example is that parts of light calculations that can be omitted for points far from the light source.

These strategies are valid for the current GPU generation, and are expected to be valid also in the future.

4.2 High level languages and drivers

On microprocessors high-level programs are compiled to assembly code closely related to the instruction set of the microprocessor family. The microprocessor instruction sets are valid for many years, and can be extended in later generations of the microprocessor family.

As current GPUs are targeted at the game market, the need for backward compatibility is different from microprocessors. GPUs are programmed in high level languages, and the GPU manufacturers supply drivers that map the shader program on to the hardware. Although assembly-like languages exist for families of GPUs, they are not recommended to be used for programming. However, they are a good tool for understanding how compiled shader code is mapped onto hardware resources. By studying the assembly-like code generated, one can learn how to restructure the fragment shader for more optimized use of resources. As an example we used this approach when more registers than necessary were allocated in a fragment shader. By looking at the assembly-like code, we realized how to restructure the fragment shader to reduce the number of registers allocated.

4.2.1 GPU high level languages

GPUs have the great advantage that there exist widespread, platform- and system-independent graphics APIs: DirectX and OpenGL. The parallel processing power of GPUs can be accessed via corresponding high-level languages:

- HLSL (DirectX High-Level Shading Language), a C-like language that is part of DirectX from Microsoft, developed in cooperation with nVidia. Microsoft is actively influencing GPU development by requiring new generations of GPUs to be compliant with the evolution of DirectX [4].

- GLSL (OpenGL Shading Language), a C-like programming language independent of the operating system. The major GPU manufacturers make their innovations available through OpenGL extensions to promote their innovation for the OpenGL standard [5].
- Cg (C for graphics), another C-like high-level language developed by nVidia for nVidia GPUs. Cg is independent of the graphics API and the compiler can generate shader code for OpenGL and DirectX. This independence has allowed a faster introduction of higher abstractions into the language [1].

4.3 Writing and debugging shaders

For those who have limited experience in graphics programming, and limited knowledge on the graphics pipeline, writing shaders just based on documentation is currently hard. Our experience is that the best for novices is to start from an existing application that is proved to be working correctly and gradually modify this to reflect the functionality you want your new application to have. When you detect that a modification to an existing shader does not work as you expect, the reasons can be numerous:

- You may have an error in your algorithm.
- You may have misunderstood the functionality of the shader language.
- The driver for the GPU is not working properly.

As there are currently no normal debuggers for GPUs, debugging has to be based on reading values from textures and analyzing these. To assist in such manual debugging, visualizing the values of computational grid as an image can be helpful for understanding the behavior of the shader. When the GPU is used as a “for loop” accelerator as addressed in Section 5.4, comparing the combined CPU/GPU implementation and a CPU-implementation can be of great value, both for debugging and for documenting the speedup achieved.

4.4 GPU-information on the web

To get an overview of what is going on world-wide with respect to use of the GPU as a computational resource, consult the GPGPU-pages, “*General-Purpose Computation Using Graphics Hardware*”, http://www.gpgpu.org/.

When we started our GPU-project these pages were a central resource for information. These web-pages are frequently updated, thus the general evolution of GPU-technology and applications can be followed in these pages.

5. Programming the fragment processor

To understand how the GPU can be used as a computational resource we have to understand how the graphics pipeline is used for rendering. A simplified view of the programmable graphics pipeline is depicted in Figure 1:

- First geometry has to be defined and the information to be processed attached to the geometry.
- The geometric structures are assembled to primitives.
- The primitives are then clipped to remove segments outside of the defined window.
- The geometric primitives are then rasterized according to the resolution of the render target. In this process the fragments to be processed by the fragment processor are produced.
- Then for each fragment the fragment shader can be executed.

Another important key to GPU-programming is the role of textures. A 2D texture is a rectangular image of dimension n×m. The graphics card offers two ways of accessing a texture, either through integer indices, or through normalized texture coordinates. To avoid knowing the exact dimensions of a texture when reading information from it, normalized floating point texture coordinates [0,1]× [0,1] have been introduced.

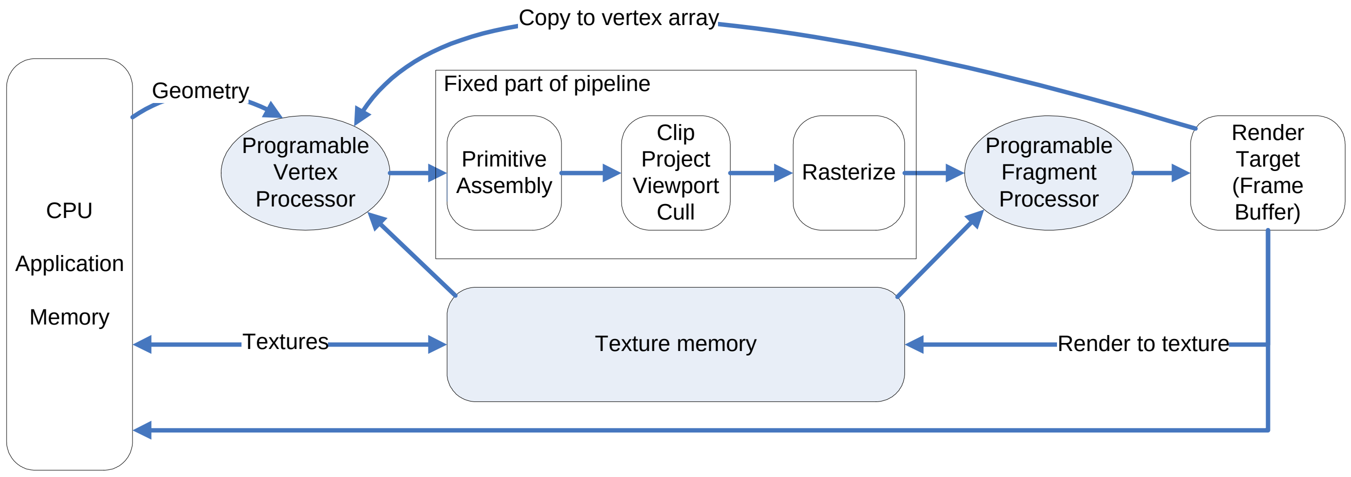

Figure 1. A simplified view of the graphics pipeline and the programmable vertex and fragment processors. Note that for the fragment processor to be triggered, all the steps before the fragment processor in the rendering pipeline have to be executed. Thus a geometry has to “own” the data to be executed on the fragment processor; this geometry has to survive the clipping and culling, and the rasterizer has to produce the proper fragments.

5.1 Default geometry and texture coordinates

To be able to execute programs aiming at non graphical problems on the fragment processor, default geometry and the visible region of space has to be defined. The following definition ensures that we get a match between the (x,y) coordinates of the geometry and the texture coordinates:

- Define a rectangle (quad) with corners: (0,0,0), (1,0,0), (1,1,0) and (0,1,0).
- Assign texture coordinates (0,0), (1,0), (1,1) and (0,1) to the respective corners of the rectangle.
- Define the viewing volume such that the quad defined above just covers the window.

This will trigger execution of all fragments with texture coordinates in the rectangle (0,0), (1,0), (1,1) and (0,1).

The above set up is conceptually the simplest. However, a rectangle is broken into two triangles in the primitive assembly process. The GPU prefetches texture data close to the current fragment to speedup fragment execution. However, this prefetch is only from the current geometric primitive being rasterized. If the fragment program needs to access texture data not associated with the current primitive, the program will have to wait for the texture data to be fetched. By making one larger triangle this latency can be avoided:

- Define a triangle with corners: (0,0,0), (2,0,0), (0,2,0).
- Assign texture coordinates (0,0), (2,0), and (0,2) to the respective corners of the triangle.
- Define the viewing volume as described above, as in the case of the quad.

The fragments outside the rectangle (0,0), (1,0), (1,1) and (0,1) will be removed by the clipping stage in the graphics pipeline, and the fragments in the rectangle (0,0), (1,0), (1,1) and (0,1) will be executed by the fragment processor. The fragment processing will be associated with only one triangle and not two triangles. This setup makes textures and render targets similar to two dimensional arrays, however, with the exception that the elements are addressed with floating point numbers in the interval [0,1] rather than integer indices.

5.2 Fragment shaders: Read only from textures; Write only to render targets

On current GPUs fragment programs can read from many textures but write to currently a maximum of 4 render targets. As we do not have control of the sequence in which the fragments are executed, algorithms such as Gaussian elimination, which combine writing of temporary data and final data in one matrix, will have to be executed as a series of shaders. The execution of a new shader is initiated by a new default geometry followed by the other steps in the graphics pipeline. At the start of 2005 typically 8000 shaders a second can be executed on a GeForce Series 6 GPU.

To benefit from the computing power of the fragment shader each fragment shader has to be computational heavy, or else the initiation process for new shaders will take too much time, and possibly also be slowed down by the maximum of shaders that can be executed in a second.

5.3 Using results from a fragment shader

The arrows in Figures 1 and 2 illustrate that the results of a fragment program can be:

- Copied to a vertex array and input to a subsequent vertex shader.
- Converted to a texture and used as input to a subsequent fragment shader.
- Be returned to the application.
- Visualized.

Although the application has to initiate a new shader for the next step of processing of results from a shader, the actual results can reside in the GPU. The execution of a series of shaders to perform a given task will not be slowed down by bandwidth between the CPU and GPU provided that the results can reside in the GPU. However, if the CPU has to process the data before the next shader can be initiated, the bandwidth will soon become a bottleneck.

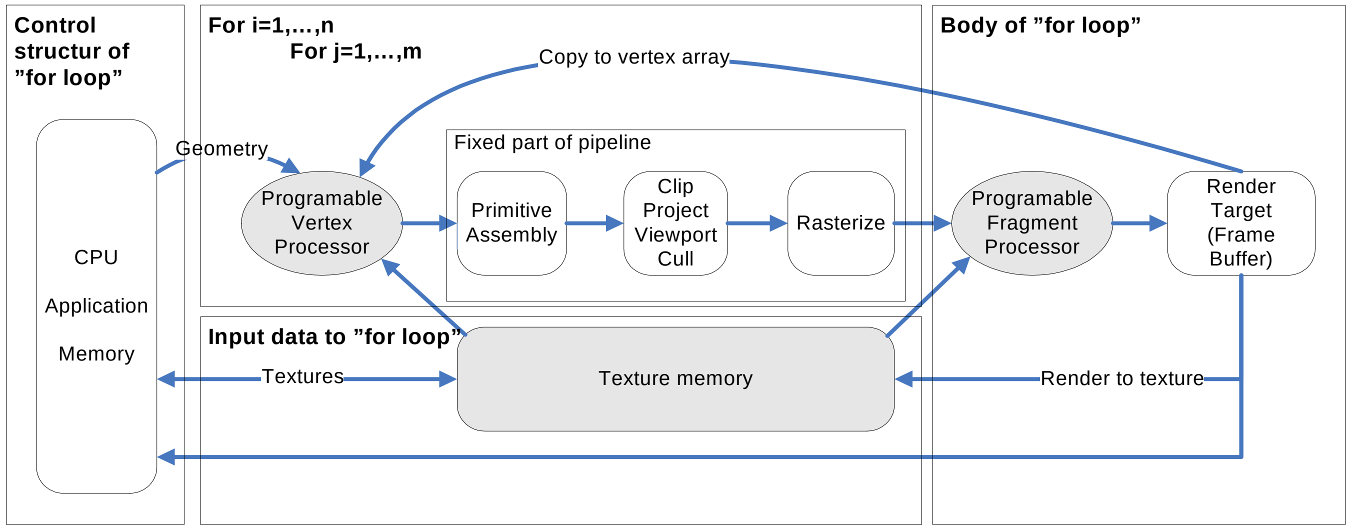

Figure 2. Loop structure implemented on the fragment processor.

5.4 “For loops” and the fragment processor

With the setup for addressing textures described in Section 5.1 the implementation of two nested “for loops” on the fragment processor is straightforward.

Looking at Figure 2 we see that the control structure of the loop:

```
For i=1,…,n;
    For j=1,…,m;
```

is replaced by calls to the graphics package. For OpenGL this can look like :

- Initiation of a viewport

```
glViewport( 0, 0, n, m);
```

- Initiation of a quad

```
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f(0.0f, 1.0f, 0.0f);
glEnd();
```

- Definition of the clipping volume such that the quad is just inside the viewing volume.

This setup will ensure that the rasterizer makes n×m fragments with texture coordinates in the domain [0,1]×[0,1].

The body of the loop will be the fragment shader, while input data is read from texture memory and results are written to the render target.

6. Examples of GPU implementations

We illustrate in Sections 6.1 and 6.2 how simple the implementation of shaders for the solution of finite difference methods for partial differential equations can be. More advanced equations are addressed in [2]. Section 6.3 addresses adaptive tessellation of parametric surfaces, where a complex algorithm is split between the CPU and the GPU.

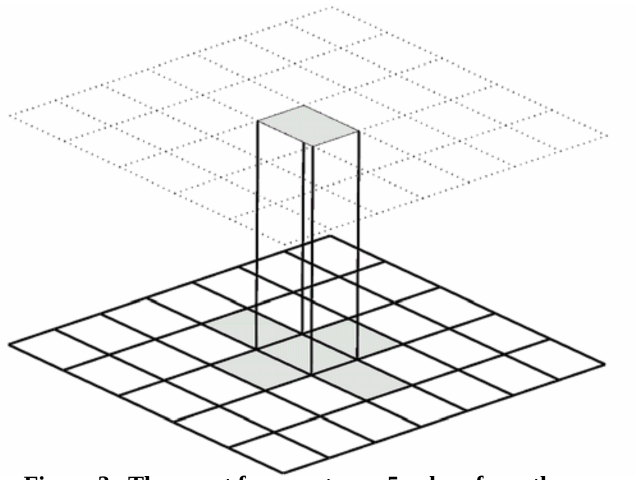
Figure 3. Then next fragment uses 5 values from the texture representing the previous time when solving the heat equation using a forward difference scheme.

6.1 Heat equation

The 2D heat equation describes transport of heat in a body

$$u_t = u_{xx} + u_{yy}\,.$$

It is an example of a simple partial differential equation. Discretication of the equation by finite differences over a regular grid gives the following scheme

$$U_{i,j}^{n+1} = U_{i,j}^{n} + \frac{k}{h^2}\left(U_{i+1,j}^{n} + U_{i-1,j}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n} - 4U_{i,j}^{n}\right).$$

The values at a given time step are calculated by combining five values at the previous time step, see Figure 3. This is well suited as a fragment program, see Listing 1, as each time step can be performed by a shader, and the results of one time step is the basis for calculating the values at the next time step.

```
[Heat Equation Vertex shader]

varying vec4 texXcoord;
varying vec4 texYcoord;
uniform vec2 dXY;

void main(void)
{
  texXcoord=gl_MultiTexCoord0.yxxx +
              vec4(0.0,0.0,-1.0,1.0)*dXY.x;
  texYcoord=gl_MultiTexCoord0.xyyy +
              vec4(0.0,0.0,-1.0,1.0)*dXY.y;

  gl_Position = gl_ModelViewProjectionMatrix
                                *gl_Vertex;
}
```

```
[Heat Equation Fragment shader]

varying vec4 texXcoord;
varying vec4 texYcoord;

uniform sampler2D heatTex;
uniform float r;

void main(void)
{
  vec4 col;
  vec4 tex = texture2D(heatTex, texXcoord.yx);
  vec4 tex0 = texture2D(heatTex, texXcoord.zx);
  vec4 tex1 = texture2D(heatTex, texXcoord.wx);
  vec4 tex2 = texture2D(heatTex, texYcoord.xz);
  vec4 tex3 = texture2D(heatTex, texYcoord.xw);

  col = tex + r*(tex0+tex1-4.0*tex+tex2+tex3);
  gl_FragColor = vec4(col);
}
```

Listing 1. Fragment and vertex shaders for solving the heat equation using a forward difference scheme.

6.2 Linear Wave Equation

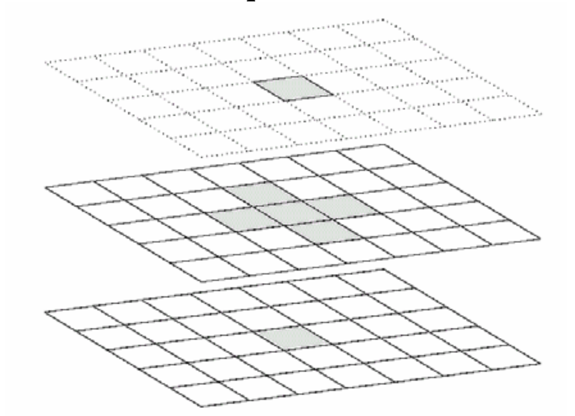

Figure 4. The fragment at time step (n+1) uses five values from the texture representing the nth time step and one value from the texture for the (n-1)th time step when solving the linear wave equation using an explicit difference scheme.

The partial differential equation for linear waves is described by

$$u_{tt} = u_{xx} + u_{yy} .$$

Since there are second derivatives also in time, the centered difference scheme will use values from two previous time steps.

$$U_{i,j}^{n+1} = 2U_{i,j}^{n} - U_{i,j}^{n-1} + \frac{k}{h^2}\left(U_{i+1,j}^{n} + U_{i-1,j}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n} - 4U_{i,j}^{n}\right)$$

Here the values at time steps (n+1) are calculated from 5 values of the nth time step, and one value of (n-1)th time step, see Figure 4, The implementation is well suited by implementation as a fragment shader, see Listing 2.

6.3 Adaptive tessellation of parametric surfaces

The above examples show fairly simple schemes for solving partial differential equations on a rectangular domain. We have also addressed real time surface visualization aiming at display qualities enabled by sharing the work load of the algorithm between the CPU and programmable GPU.

In [3] a method for providing high quality view-dependent tessellations of certain types of parametric surfaces including B-spline surfaces is described. The objective is to ensure that, regardless of the camera positions, the rendered triangles will cover approximately the same number of pixels. In order to do this efficiently we use the GPU to sample and evaluate the surfaces.

7. ACKNOWLEDGMENTS

We thank the Research Council of Norway for funding the project *"Graphics hardware as a high-end computational resource"*, through the contract 158911/I30.

8. REFERENCES

[1] R. Fernando and M.J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] T.R. Hagen, Henriksen M.O., Hjelmervik J. M., and Lie K.A., *How to Solve Systems of Conservation Laws Numerically Using the Graphics Processor as a High-Performance Computational Engine*, submitted for publication in "*Geometric Modelling, Numerical Simulation and Optimization: Industrial Mathematics at SINTEF*", Springer

[3] J. M. Hjelmervik and Hagen T. R., *GPU-Based Screen Space Tessellation*, in *Mathematical Methods for Curves and Surfaces: Tromsø, 2004,* M. Dæhlen, K. Mørken, and L. L. Schumaker (eds.), Nashboro Press, 2005.

[4] Microsoft, *DirectX: multimedia application programming interfaces*. http://www.microsoft.com/windows/directx/default.aspx.

[5] R.J. Rost, *OpenGL(R) Shading Language*, Addison-Wesley, 2004.

```
[Linear Wave Equation Vertex shader]

varying vec4 texXcoord;
varying vec4 texYcoord;
uniform vec2 dXY;

void main(void)
{
  texXcoord=gl_MultiTexCoord0.yxxx +
                     vec4(0.0,0.0,-1.0,1.0)*dXY.x;
  texYcoord=gl_MultiTexCoord0.xyyy +
                     vec4(0.0,0.0,-1.0,1.0)*dXY.y;

  gl_Position = gl_ModelViewProjectionMatrix *
                                        gl_Vertex;
}
```

```
[Linear Wave Equation Fragment shader]
varying vec4 texXcoord;
varying vec4 texYcoord;

uniform sampler2D texCurrent;
uniform sampler2D texLast;

void main(void)
{
  vec4 col;
  vec4 tex = texture2D(texCurrent, texXcoord.yx);
  vec4 tex0 = texture2D(texCurrent,texXcoord.wx);
  vec4 tex1 = texture2D(texCurrent,texXcoord.zx);
  vec4 tex2 = texture2D(texCurrent,texYcoord.xw);
  vec4 tex3 = texture2D(texCurrent,texYcoord.xz);
  vec4 texL = texture2D(texLast, texXcoord.yx);

  gl_FragColor = (2.0 * tex - texL + (0.5)*(tex0
          + tex1 + tex2 + tex3 - 4.0*tex))*0.92;
}
```

Listing 2. Fragment and vertex shaders for solving the linear wave equation using a forward difference scheme.