University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitní 8
306 14 Pilsen
Czech Republic

The GPU and Graphic Algorithms

State of the Art and Concept of PhD Thesis

Ivo Hanák

Technical Report No. DCSE/TR-*2005-05*
April, *2005*

Distribution: public

Technical Report No. DSCE/TR-*2005-05*
April 2005

The GPU and Graphic Algorithms

Ivo Hanák

Abstract

In the begging, graphic hardware was utilized only to convert binary information to display device. Through its evolution, it gained capabilities and last improvement enhanced it so far that it can be considered as a next processing unit. Thus, this document concerns with graphic hardware, its applications and proposed improvements. It is an overview with selected samples that shall introduce the reader to the area of research a application of graphic hardware.

This work was supported by Microsoft Research Ltd. project FPG-PLT-511568, MSM 23500005 „Záměry", Ati Technologies Inc., and project 3DTV: Integrated Three-Dimensional Television, NoE.
Copies of this report are available on
http://www.kiv.zcu.cz/publications
or by surface mail on request sent to the following address:

University of West Bohemia
Department of Computer Science and Engineering
Univerzitni 8
306 14 Pilsen
Czech Republic

Acknowledgments

I would like also to thank colleagues at University of West Bohemia, Faculty of Applied Sciences, Department of Informatics for giving me interesting ideas and for consultations on the topic, as well. Finally, I would like to thank Ati for providing us with hardware that was used in experiments.

Table of Contents

1 Introduction

Computer graphics is an area of computer science aimed mostly on image synthesis, i.e. its purpose is to generate artificial images based on scene description. This is true, even thought some branches of computer graphics performs computation to produce either different representation of scene or modified scene with requested attributes. However, at the end, the generated data are utilized in process of image synthesis to find a balance between visual realism and computational costs.

In its beginning, the computer graphics was able to provide only simple, low-resolution and usually line based images on displays of various hardware principles. In these day the computational costs were quite high in comparison to visual quality of resulting images. Thus, algorithms has to be invented that improves overall performance. Nevertheless, that was not enough because as the computational power of computers was increasing so was increasing the resolution of displays and the complexity of displayed scenes.

The importing thing in computer graphics image synthesis is that the display hardware structure allows or even requires an image to be composed of N-dimensional array of image element, i.e. **pixels**. The constant N depends on display construction and in the case of common 2D color display, the N is equal to three. The task of image synthesis is then a question of value estimation for every pixel in the image, i.e. every pixel has to be processed in order to construct the resulting image.

There are many approaches that in the end lead to a value of the pixel. Some of these approaches provide convincing image others provide just crude approximation of the scene. Most of these approaches consist of large number of rather simple steps that are repeated during image construction many times. And thats a point where hardware enters the scene.

The hardware was there from the beginning of the computers. Without the hardware there would be no software because software is just an attempt to generalize it and therefore both simplify and speedup the process of new solution creation. Still, it is true that software solution is much slower than its hardware equivalent. In some cases, solved problem leads to large number of similar simple steps whose computation consumes significant amount of central processor unit (CPU) time. In such case a hardware performance overweights benefits of software solution.

In the field of computer graphics there exist many experimental and rather specialized pieces of hardware whose only purpose is to improve overall running time of several algorithms. In majority of cases these were rather experimental platforms and thus many of these solution were abandoned due to their ineffectiveness in comparison of computer science advances, some of the found its way to an industrial utilization.

In the industrial area, hardware dedicated to aid image synthesis helps computer aided design (CAD) to visualize models described by its surface with high detail. Such hardware is able to perform a task that is both significant in process of image synthesis and rather time consuming,

i.e. the task of rendered primitive decomposing to elements such as pixels including a simple port-processing. This decreases the load on the CPU and thus allows visualization of large scenes at interactive framerates. This is rather satisfying for CAD purposes.

However, it was not enough for game industry. Thanks to the fact that this dedicated hardware became available for the home users, the games began to utilize it in order to provide higher realism and/or more complicated environments. It was the game industry that later on supplied a stimuli for enhancement of graphic hardware.

In first generations the hardware was able to perform triangle rasterization including texture mapping, occlusion solution, and other pixel post-processing tasks. The solution was hard-wired with a possibility for limited customization and/or programmability. Thanks to its limitations, the hardware was able to outperform general CPUs and that was a point were attempts were made to utilize such graphic hardware for little bit different purposes than the image synthesis.

Later on, the rest of the scene processing became a part of hardware allowing a game developers to move almost complete visualization load out of the CPU and thus gain more time for other aspects of the game. However, still there were tasks concerning scene visualization that had to be computed on the CPU as well as visual effects that were hard to cope on that hardware.

Therefore, a possibility of pipeline customization by application was added to the list of graphic hardware capabilities. The next logical step was then to allow a user to gain almost complete control over particular components of the pipeline by execution of customizable programs that replace the hardwired solution.

The original purpose for many of these enhancement was to provide an environment for special effects that would improve both quality and realism of the resulting image in real-time but not decrease performance significantly at the same time. Thanks to that the graphic hardware came from state of dedicated single purpose hardware to state of processing unit able to perform specialized but fast numerical operation. Thus, the graphics processing unit (GPU) could be considered as a hi-performance special purpose processing unit running in parallel to the CPU.

This document concerns the graphic hardware and its capabilities including examples of possible hardware-aided solutions. It is mostly aimed on the GPU because this platform is widely available and therefore its application may be utilized by wide public. The document consists of two major parts: the state of the art in the graphic hardware computing and future work in a field of the graphic hardware. The author assumes a reader has at least a basic knowledge of C language syntax and computer graphic principle basics.

2 Hardware Architectures

This chapter contains introduction into hardware itself. It divides hardware by its availability for home user. It also describes capabilities and important aspects of these hardware if relevant for contents of this work.

2.1 Classical Pipeline

At the begging of graphic hardware evolution there was a need to perform image synthesis with a high performance. The process was based on displaying of triangles on raster displays. This was simple enough to be implemented straightforwardly in the hardware. The resulting hardware was enhanced by additional capability during its evolution until it contained complete pipeline required for rasterization of a scene described by surface in a form of triangles.

The pipeline that is description of data flow in current common widely available hardware will be called as **classical pipeline** because it is the only one pipeline that survived its evolution and found its place in both industrial and home use. Also, majority of approaches described in this work utilizes it directly and/or derives from it, hence the denotation. The implemented pipeline can be divided into two major blocks: the vertex pipeline and the pixel pipeline [DX05].

2.1.1 Vertex Pipeline

The vertex pipeline prepares a geometry for rasterization (see Figure 2.1). The input of this pipeline consists of data provided by a user. In majority of cases those are rendering primitives or their modification such as strips and fans that may be used in order to decrease the overall storage requirements and bus traffic. In a special case the input may consist of high-order surfaces.

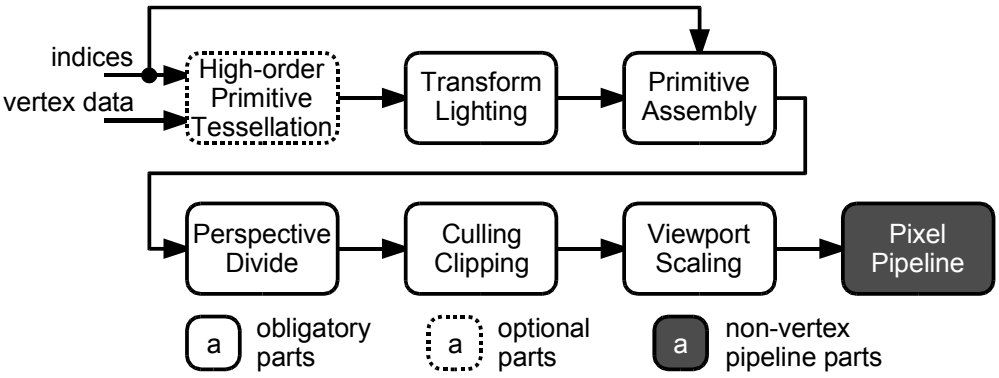


Figure 2.1: Vertex pipeline scheme.

The first step of the vertex pipeline is a **tessellation** of **high-order primitives**. This step is optional[1] and if present, it is preformed only when either a high-order primitive is passed or the tessellation functionality is enabled. The output of the tessellation step are triangles whose number is usually larger than a number of incoming ones. The benefit of the approach is to decrease of both data storage and data transfer because all operations are performed on chip. Currently, there are two possible high-order primitives:

- **RT-Patches**[2] that provide hardware support for high-order primitives. The functionality is accessible through a special set of functions and allows the user to utilize Bézier, B-spline, and Catmull-Rom patches. The tessellation does not take any quality measurement into account and utilizes user provided maximum depth of division. The depth controls dividing of the patch into a set of triangles. For more details, refer to [Mor99, DX05].
- **N-Patches** or **PN Triangles**[3] that perform a conversion from a triangle with normals at vertices to a triangular Bézier surface according to an interpolation rules. After the conversion the user supplied maximum depth is used to divide the patch back into triangles.

  This functionality is rather transparent for a user because it is performed automatically, if enabled. N-patches provide surface smoothing and together with displacement maps (see below) they allow displaying of models of various level of detail (LOD) without additional structures and/or storage requirement. For more details, refer to [Vla01, DX05].

As the next, **transformation** and **lighting** computation takes place. The goal of this step is to prepare vertices for clipping by a viewing frustum. This is done by transforming of vertices into a unit clip space with camera at origin of the space oriented along the z-axis[4]. The lighting is a local illumination system that utilizes the Phong lighting model [Wat99, Pho75] in order to compute lighting for every vertex.

Afterwards, **primitives** are **assembled**. This step is required because the input data my not be configured as a list of vertices that forms a desired primitive. Based on performance issues the user is able to pass a list of primitives where a next subsequent primitive utilizes an information for the previous one thus saving the space for storing. Besides that, there are indexed primitives where the user passes an array of vertices and an array of indices that point to the vertex array and thus form the desired primitive.

The primitive assembling step utilized a buffer that hold processed vertices before their embedding to a given primitive. Surely, this buffer is a rather limited in its size and therefore the pipeline may throw some processed vertices if an input stream exhibits inappropriate order. This may cause a performance drop down.

Afterwards, perspective divide, culling, and clipping takes a place. **Perspective divide** transforms vertices from P^3 a projective extension of E^3 space back to the E^3 space. **Culling** then removes triangles that are facing away from the viewer, if enabled. It assumes that triangles facing away from user and on reversed side of closed solid object and thus their rasterization would have not impact on the resulting image.

The **clipping** on the other hand clips all elements and/or its parts that are outside the viewing frustum, i.e. unit clipping space/cube. If a primitive is partially inside the cube, the mechanism

1 Currently only Ati-based GPUs has this capability widely supported.
2 RT-patches were implemented first by nVidia.
3 N-patches were implemented first by Ati
4 For the OpenGL, the camera is oriented towards negative z-axis. For the Direct3D, the camera is oriented towards positive z-axis, i.e. it is left-haded system.

computes clipped coordinates. For a triangle this can lead to a hexagon that is divided back to a list of triangles because the rasterizer is able to process triangles only.

Actual algorithm utilized for clipping is hardware provider dependent and is part of an internal non-public structure of the hardware. From capabilities and runtime requirements it seems that the clipping algorithm could be a Sutherland-Hodgman or its modification [Wat99, Ska04].

Together with the clipping another term could be mentioned: a **guard-band clipping** [Die99]. This feature is not very important in current hardware and it is rather part of pixel pipeline. Nevertheless it has impact on the vertex pipeline construction and thus it is mentioned here. Guard band clipping is aimed on decreasing of the computational load on a vertex pipeline by moving it towards the high-performance rasterized in the pixel pipeline.

The idea is based on a fact that there a lot of rather small triangles that are incident with edges of the unit clip space. Such triangles can be lead to a rasterizer, which automatically discards out-of-screen pixels. It is sure that rasterizer works only on specified limited area[5] and therefore it may happen that given triangle is even outside the enlarged workspace of the rasterizer. Such triangle has to be clipped before the rasterization.

Nevertheless, thanks to the guard-band clipping a lot of triangles does not need to be clipped at all before entering the rasterizer. This enhancement has significant impact for older hardware graphic hardware mostly that lacks implementation of the vertex pipeline. In such case, the guard-band clipping utilization decreases load on the CPU.

The last step performed on the vertex pipeline is the window-viewport transformation that converts projected triangles from unit screen to an actual screen size. The output of this step is the output of the vertex pipeline and it is directly processed by the rasterizer.

2.1.2 Pixel Pipeline

The pixel pipeline is the oldest part of the pipeline implementation and its goal is a rasterization of the triangle and computation of a color for particular generated pixel, see Figure 2.2. The input of the pipeline are clipped triangles transformed to screen coordinates.

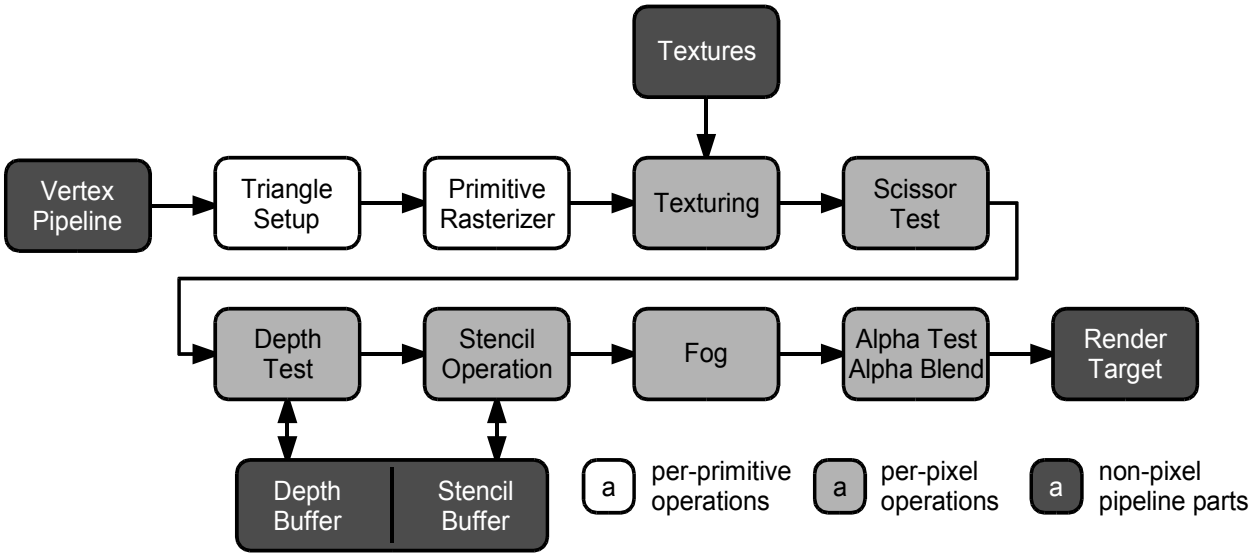


Figure 2.2: Pixel pipeline scheme.

5 Size of area depends on particular version of the chip. In the case of nVidia cards, this area is slightly larger than the rest of hardware providers.

The major part of the pixel pipeline consist of the **rasterizer** that decomposes given triangle to pixels[6] together with proper perspective correct [Cat80, Wat99] interpolation of a vertex associated information, i.e. texture coordinates, color, depth information, fog, etc. The rasterizer utilizes a scan-line conversion algorithm with triangle prepares in the **triangle setup** phase. For a hardware implementation, there are two possible mechanisms that both lead to rasterized triangle as described in [Mit00]:

- Algorithm based on linear interpolation that first forms vertical slopes from the triangle during the triangle setup and then utilizes these slopes to create a set of horizontal pixel segments. Each such segment is iterated in order to generate individual pixels.
- Algorithm based on edge function handles the triangles as a set of linear edge functions that are utilized to check whether given pixel is inside the triangle thus being processed or whether is outside the triangle thus being omitted. The approach can be improved by utilizing a rectangular array of pixels denoted as pixelstamp: processing of each pixelstamp may run in parallel.

Then, a **texture mapping** is performed for each generated pixel. The common structure that is used for texture mapping is a cascade-like connection of texture units that utilizes texture coordinates associated with the pixel. The number of texture units is strictly hardware dependent; in a majority of cases the number is 2—3. This number is satisfying for the most of application such as games because it allows mapping of a surface texture and a light map in a single pass. The output of this step is a pixel with associated color, fog, and depth information.

Later on, a post-processing is performed on the pixel. This **post-processing** allows a significant amount of customization and thus enables the user to create several interesting visual effects. The pixel is tested against scissors rectangle, a depth buffer, and a **stencil buffer** [Kil99] that is in the most of cases a part of the depth buffer. During the stencil test, the pixel is compared against the depth buffer and the results of both depth and stencil test determines fate of the pixel[7].

Last post-processing operation involves application of a fog based on a fog value associated with the pixel. Afterwards, an alpha-blending is performed according to the pipeline setup and a pixel associated alpha value. This includes a simple alpha test that may discard pixel with too low alpha in order to render an image with only a minimal number of visual artifacts utilizing depth buffer only. Then, the result is written to the current render target, also denoted as color buffer.

2.2 Programmable Classical Pipeline

The programmable pipeline is a next evolution step in commodity graphic hardware development. The basic idea for such enhancement was to allow an application to perform customizable texture mapping in order to improve visual quality of synthesized image without significant impact on the performance.

At the first, enhancement were made in vertex pipeline by allowing a simple program to be executed for modification of vertex blending and/or texture coordinate generation. This was possible because the vertex pipeline has lower requirements on a throughput in a comparison to the pixel one.

6 In some literature the value in this stage is called a fragment rather than a pixel because of the the contains more additional information besides its color.

7 The pixel may be either passed through or discarded. The corresponding value in the stencil buffer may be updated as well.

As the next, the pixel pipeline was enhanced by allowing an application to customize texture blending. This modification significantly enlarged the range of possible effect for real-time synthesized graphics and it was known as a **register combiner** [Kil04]. This was the last step before a programmability of both pixel and vertex pipeline became a reality [Lin01].

Surely, not all parts of the pipeline are programmable. However, the hard-wired parts usually provides functionality that is common for all image synthesis tasks and thus its programmability would not mean almost any benefit besides the increase in both computation time and cost due to programmability.

The major idea behind the graphic hardware development in this area is to keep it as simple as possible in order to gain maximum performance. Therefore, the pipeline still retains its pipeline characteristics thus resulting to several limitation for programmable blocks in comparison to the CPU. However, thanks to these limitation the pipeline may apply massive parallelization in both number of pipelines and number of operands processed by a single instruction and thus it outperforms comparable CPU.

Currently, the pipeline contains two programmable blocks: the vertex shader and the pixel shader. In some environments such as OpenGL the pixel shader is usually called a **fragment shader**. This difference is based on a fact that while the pixel shader output is a pixel color, the input is a pixel with additional information, i.e. a fragment of a rasterized surface. In such case, term pixel shader denotes a post-pixel production [3DL04]. However, for sake of this work uniformity, the therm pixel shader origination from Direct3D [DX05] is used.

2.2.1 Common Attributes

As it was mentioned in previous paragraph, the pipeline has two programmable blocks. Even thought these blocks are situated at different places in the pipeline and provide different output at the same time, both follow similar rules and have similar capabilities and limitations as well.

The most important fact that every user should be aware of is the **locality** of both shaders. Neither the pixel shader nor the vertex shader are aware of processed element context during its processing. This means that the user is not able to access neighborhood in both geometrical and image meaning because the shader has no information about it.

On the other hand, this restriction allows simple parallelization of element processing thanks to the lack of synchronization for read/write request that may appear from within the shader code. Similar to the rule already mentioned, the shader is unable to alter the context of the element, e.g. the user is not allowed to convert triangle to points or to modify position of vertex within the triangle.

The next important attribute of the shader is its execution: shader is **executed upon data availability**. The user is not able to execute the shader separately without any supplied data. This limitation is based on a fact that shaders are part of hard-wired pipeline and therefore the capability of execution independently on pipeline would disturb its structures and complicate its hardware implementation.

Similar to that, the next important restriction in shader programming is that the shader is **not capable to generate** any additional elements. It is sure that the shader is capable of generating new data upon given one but those data has to be associated with currently processed element. The capability of new element generation would lead to a complication and therefore a performance drop down for the whole pipeline.

In previous paragraphs, the execution management and input/output capabilities were mentioned. However, still there is a one important area that has restriction applied as well: the **memory management**. The important is, the shader has no memory management at all. It is not capable to access memory of the graphic adapter directly and random manner.

The only area available to the shader code for random read/write access is a shader local memory that is limited in its size. The size of the memory has no dependence on memory available on the graphic board because it is implemented as a set of read/write on-chip registers and this is very small and very fast at the same time.

Memory substituting registers as freely accessible to the shader code but their values are undefined once the shader finishes processing particular element, i.e. the memory is intended only for a temporary storage. This means that all computations in shaders are **stateless** and the only way of obtaining of computed results is through the output of the pipeline, i.e. the render target and/or depth buffer.

Besides limitation for computation and memory management, **available data types** are also important feature of the shader. The native type supported by a majority of GPUs is a 32-bit floating-point **single** [IEE85], i.e. **float** in C-language syntax. However, there exist other floating-point formats that were mostly added due to either performance or construction reasons: half-precision floating-point and fixed-point data type.

The half-precision floating-point data type is usually denoted as a **half**. Its length is 16 bits and was added due to performance reasons in order to sacrifice accuracy and gain performance at the same time. It is recommended for a shader code to use this data type whenever the accuracy is not crucial such as color computation intended for displaying.

On the other hand, the fixed-point data type was added due to construction reasons. It is usually denoted as **fixed** and its length is usually 12 bits. It is the only numerical data type available on first generation of graphic hardware with shader model of version less than 2.0. This is because older hardware utilize faster but less accurate fixed-point point arithmetic. It has an appropriate accuracy for image synthesis purposes and is much faster in comparison to the floating-point arithmetic[8].

However, according the nature of fixed-point computations, the accuracy is very restricted and the range of values is strictly given. Thus, all mathematic operations on the hardware with shader model less than 2.0 including fixed data type are **saturated** to a range $\langle -1;1 \rangle$. This makes the older hardware almost useless for general purpose computing but it is sufficient for basic image synthesis computations such as Phong shading.

Integers of 32-bit length are also supported but in a majority of cases it is replaced by the float internally. Besides numerical data type, the GPU is able to handle a **boolean** type that is used for flow control in a form of if-branching, both static that depends on values supplied by the application and dynamic that depends on values computed in the shader code.

The last available data type are **samplers**. These are intended for obtaining a color sample from a texture and their behavior is determined by the pipeline settings. Currently, there are four available types of samplers that differs by the texture type they are sampling from: 1D, 2D, 3D[9], rectangle[10], and cube-map.

8 Try to compare implementation of floating-point and fixed-point division of two numbers.

9 3D Texture is basically a volume texture.

10 Rectangle texture is 2D texture with a side not equal to 2^k, $k \in \mathbb{N}$. On some hardware, rectangle texture may have limited set of possible filters for obtaining a sample.

Value sampled from a texture is always a four-component vector of floating-point number no matter what is the actual data type of a texture element, i.e. **texel**. Texels with color components of integer data types are converted to a range of $\langle 0;1 \rangle$. Floating-point textures are sampled without any value modification and/or saturation.

Thanks to fact that in mathematics for computer graphics the most used one computational elements are vectors, the GPU is able to operate with an up to four-component vectors instead of plain scalars. The important attribute of the modification is a capability to convert between vectors of the same type but various lengths.

From the code point of view, the vector is handled similar to a structure, i.e. individual elements are accessed via point operator. There are three sets of names for element: `xyzw`, `rgba`, and `stpq`. Event thought it is not allowed to mix sets used for and while accessing a single particular register, they do not distinguish in functionality. The reason for their existence is an attempt to improve readability of the code mostly when done in the assembly language, see Chapter 3.

The next important capability concerning numerical data types is a possibility to perform user specified vector component rearrangement. This basically allows the user to select particular elements of a vector to be used either as source or target for an operation, e.g. consider four-component vector $\boldsymbol{v}$ than code `v.zx` denotes a two-component vector that has first component of value v_z and second value of v_x. When used for a argument of a function, this operation is called a **swizzling** and when used for specification of target vector element it is called a **masking**.

Similar to that, vector per-component negation is supported, too. All mentioned operation with registers are performed naturally without any requirements for special instruction. Therefore, this modification allows a shader to perform several computation with both color and position in P^3 the projective 3D space without additional computational and/or shader instruction requirements. Besides that, each register has access restriction applied. These restriction divides all available registers into three groups. Actual numbers of registers depends on particular shader model and shader type. For details, refer to Appendix A. Groups are:

- **Read-only** registers. These are registers that are considered as an input to the shader. First of all, there are input from a pipeline. The number of these registers as well as their semantics depends on particular shader and is described in next sections.

  Next, there are numerical and boolean **constants**: a special relatively large set of registers containing values set up by the user of the shader, i.e. the application. They are often used for transferring of object and/or scene information such as transformations, light settings, material, animation information, etc. to the shader code.

  Similar to numerical constants, there are special constant registers for samplers. The exception from registers mentioned above is a **loop control register** that is read-only as well but accessible only inside of the loop and contain number of already performed loop repeats.

- **Write-only** register group that contains registers intended to hold output values that are used as input for the rest of the pipeline. Similar to the rest of registers, their numbers as well as semantics is shader type and version dependent. These registers are either floating-point vector of four component or floating-point scalar.

  The important attribute of these output registers is the fact, that some of them are obligatory. The obligatory registers have to be set prior to shader code exit. The optional ones does not require it and contains predefined value: usually zero or vector of zeros, if not set.

- **Read/write** registers supply the local memory for a shader during element processing. As it was already mentioned, such memory is strictly temporary, hence the temporary registers. Their count is low and they are usually of float data type. For exact data, refer to Appendix A. Besides numerical temporary registers, there are special purpose registers: **predicate register** that is utilized in dynamic if-branching and **address register** that is used for relative addressing of constant register, i.e. allows implementation of application supplied arrays.

The **program** that controls behavior of the shader has also several limitations. First of all, its **length** is strictly limited. In some cases the length of the program is divided by type of operation. Actual minimal numbers of instruction per shader depends on a particular shader model and are listed in a table within the Appendix B.

The next important restriction applied to a program is a limited **flow control** that is mostly understood as a branching capability. For a lower shader model, the **branching** is not available at all but it is possible to bypass it by simply computing both branches and then selecting appropriate result. This inability to perform float control simplifies hardware construction and therefore improves the overall performance of the shader [Lin01].
GPUs of newer shader models have a certain capability to perform either static or dynamic branching. The **static branching** is a situation when a condition is a boolean value supplied by the application in a form of constant, the **dynamic branching** utilizes predicate register whose value is computed during shader execution.

The **looping** is also supported and a loop exposes its control variable for read-only access as it was mentioned. The looping is static because maximum number of repeats has to be stored into a constant register. Nevertheless, when dynamic flow control is available the loop execution may be interrupted according to a condition. Maximum number of loop repeats is limited to 255.

Besides that, the next possibility for flow control are a **subroutine calls**. In the case of older hardware, this is not supported and a code of a subroutine have to be copied instead if a subroutine call is required. If this feature is available, the user may call a subroutine either statically or dynamically, i.e. it may give a condition that has to be fulfilled in order to perform the call.

Maximum **nesting depth** for subroutine calling is strictly limited and thus recursion is not available for the shader code. Nesting level for branching and looping is restricted as well. In the case of static branching, some shader models may apply restriction to a number of such instruction to appear in the code but they allow almost limitless nesting of these instruction at the same time. Nesting depth depends on shader model version and a type of the shader. For details, see Appendix B.

Operations preformed by GPU show behavior similar to Simple Instruction Multiple Data (SIMD) system in a majority of cases . They allow to process whole four-component register at once. This is one of fundamentals of the GPU computational power, i.e. parallelization on instruction level.

It is true that the overall performance of an instruction depends on a performed operation but this difference in performance when using a single scalar or when using four-element vector does not excess the coefficient of four, i.e. the instruction execution time when using a vector shall be less than four times the same but using a scalar.

Accuracy of mathematical operations utilizing floating-point value type depends on particular hardware but for shader model 3.0 and newer, the accuracy is 128 bits per register, i.e. 32 bits per single floating-point register component. For a shader model 2.0, the accuracy may be reduced to

96 bits[11] per register but this reduced accuracy should have almost any visible impact on resulting synthetic image. Lower shader models does not have support for floating-point operations at all and therefore their accuracy is limited by 12-bit fixed-point data type.

The GPU supports wide range of operations in a form of instruction. For completeness of this section a brief overview of instruction groups is provided. Supported instructions are:

- **Mathematical operations**. This group includes instruction of adding, multiplying of vectors per element, dot product that allows implementation of other important operation such as matrix multiplications, cross product, linear interpolation, rounding functions, reciprocal of scalar, reciprocal square root, logarithmic functions, and sin/cos.
- **Flow control** instruction group that contains if-branching, looping and subroutine calling, both static and dynamic.
- **Texture sampling** support including projective mapping and cube-map sampling.
- **Geometry operations** that include vector normalization, lighting computation support that shall simplify the task of disabling propagation of specular lighting if element normal has reversed direction than direction of light.

2.2.2 Vertex Shader

Vertex Shader is the first programmable block in pipeline data flow. It resides at a very beginning of the pipeline just after the tessellator. Its purpose is to process vertices and prepare them for clipping and primitive assembling. Based on its functionality, it replaces the transformation and lighting block of the vertex pipeline and whenever a vertex shader is used it has to perform all tasks done by replaced blocks.

2.2.2.1 Input

The input of the vertex shader consist of three parts:

- **Input from the pipeline** that consist of tessellator processed data. In a majority of cases these are geometrical data passed by the user to the pipeline. Data consist of up to 16 four-component vector a 32-bit floating-point per each component. Each vector has its own designated semantics, i.e. it is mapped right to a particular vertex component. None of these vectors is mandatory, but based on a fact that execution is driven by a data availability at least one of them contains valid value.
- **Constants** that are set by an application in order to modify computation performed. In a case of the vertex shader they usually consist of transformation matrices and possible light settings even thought in a majority cases, lighting is performed on per-pixel basis rather than on per-vertex bases as it is common for the hard-wired pipeline. Number of constants varies and besides four-component floating-point vectors, vectors of boolean type are allowed in order to perform static flow control.
- **Vertex textures** are the largest source of data for vertex shader and are in a majority of cases utilized for displacement mapping and for utilizing of data computed in a pixel shader during previous pass. This is rather important because it allows utilization of pipeline output in order to modify processed geometry without expensive downloading computed data back to the

11 96 bit accuracy is mostly true for graphic hardware by Ati.

CPU. Such benefit may be crucial for computation of various particle system including fluid/gas simulation.

The vertex shader is usually a place for application of the **displacement maps** [Coo84, Dog00]. These are textures that contain modification of a vertex position according to vertex texture coordinates. The modification benefits from a possibility to parametrize the surface by utilizing a texture coordinates. Based on that, let $\boldsymbol{P}(u,v)$ be a vertex of base surface, $D(u,v)$ a sample from the displacement map and $\boldsymbol{N}(u,v)$ a normalized normal of base surface then new position $\boldsymbol{P}_{new}(u,v)$ is defined as:

$$\boldsymbol{P}_{new}(u,v)=\boldsymbol{P}(u,v)+D(u,v)\boldsymbol{N}(u,v) \quad (1)$$

Vertex normal may remain unmodified even thought they are possibly incorrect for modified vertex because it is replaced by a value from a normal map during per-pixel lighting, see Section 4.1. The technique of displacement mapping is utilized whenever a silhouette modification is required.

This mechanism can be utilized together with the high-order primitives such as N-Patches in order to compute and display models with appropriate LOD on the fly. This does not save only the rendering but also storage and transport requirements because the application supplies same amount for primitives and their division into larger set of triangle is done completely on-chip.

Currently, only shader model 3.0 and newer supports vertex textures, older shader models are not capable of utilizing this input source and have to bypass it by utilizing the CPU for performing vertex data manipulations. The supported format for vertex textures is usually either a 32-bit floating-point four-component vector or 32-bit floating-point scalar per texel.

Nevertheless, in some cases vertex textures can be bypassed by a proposed extensions of the OpenGL 2.0 denoted as a pixel buffer object. From the view of this extensions all buffer objects in the video memory are basically the same and thus it is possible to utilize them as the render target, i.e. this allows to render directly to the vertex buffer and utilize such data as the geometry in the next pass of the algorithm.

2.2.2.2 Output

The output of the vertex shader is based on the pipeline requirements for vertices that has to be clipped and utilized during primitive rasterization in pixel part of the pipeline. The output itself consist of up to 12 four-component floating-point vectors. Each vector has its own semantic assigned in order to distinguish meaning of the vertex shader output. With a single exception, all output values are optional, i.e. setting of them is not required. The possible output of vertex shader are:

- **Vertex coordinate** that is the only obligatory value that has to be set up as a result of every vertex shader execution because its value is used for clipping purposes. The coordinate has to be transformed into a unit clip space and is passed out of vertex shader in homogeneous coordinates.
- **Diffuse** and **specular color** that contains colors computed during vertex lighting. These values are interpolated over the surface of rasterized primitive in perspective-correct manner and thus it is possible to estimate their appropriate value for every pixel. Interpolation is performed per element. Its meaning is mostly bound to non-programmable pixel pipeline in which they are utilized to modify the output of texture mapping. If the pixel shader is used, the meaning is user dependent but there may be a risk of saturated arithmetic on an older hardware.

- **Texture coordinates** that are utilized for transporting of texture coordinated to the pixel pipeline in order to perform texture mapping. There are up to 8 of texture coordinates that may hold almost any value possible. Similar to colors, texture coordinates are interpolated during rasterization. They are mostly used for transporting a user defined values to the pixel pipeline in order to compute several per-pixel operations, such as Phong shading where one texture coordinate contains estimation of a surface normal for given pixel.
- **Fog** and **pixel size** that are the only scalars and they are utilized by the pixel pipeline during either primitive rasterization or pixel post-processing that are both hard-wired.

2.2.2.3 Program and Operations

Operation in the vertex shader are exhibits SIMD behavior with only few exceptions and the palette of possible operation is larger than in the case of the pixel shader. The vertex shader also does not have limitation of saturated arithmetic and provides better support in a field of flow control abilities, e.g. even for shader model 2.0, static flow control in vertex shader is supported.

As it was mentioned in previous section, the vertex shader has no information concerning context of processed vertex. Whenever such information is required, the application has to pass it as a part of vertex data or in a vertex texture, if available. Also, the vertex shader is not able to generate additional geometry on its own. Fer example, refer to Subsection 4.3.2.5.

The typical use for vertex shader lies in a field of scene preparation, such as a simple transformation and per-vertex lighting. Besides that and already mentioned displacement mapping based LOD, it is possible to utilize it in a field of animation based either on simple morphing between two key meshes or on particular mathematical function, automatic texture coordinate generation, skinning of meshes, etc.

At general, the utilization of vertex shader is a field of low data throughput computation because the vertex shader is not assumed for processing of large number of elements. This assumption is based on a fact that the number of vertices in scene is significantly lower in comparison to number of generated pixels and it allows much richer functionality and higher accuracy for computations in comparison to the pixel shader.

2.2.3 Pixel Shader

The next programmable block in the pipeline is the pixel shader. It allows modification on a level of individual pixels generated during primitives rasterization. In the pipeline flow described by the Figure 2.2 it replaces texture mapping and replaces completely its functionality. It is intended to perform more sophisticated texture mapping but thanks to its capabilities it allows several effects that make hard-wired implementation of several pixel-based techniques obsolete.

2.2.3.1 Input

The input of the pixel shader consists of pipeline provided information that is basically an output of the vertex shader, samplers, and constants. Constants are similar four-component vectors as it were described in previous sections. Opposite to the vertex shader, the pixel shader constants are limited in their numbers almost to a quarter of the vertex shader count and therefore the application is not able to pass complicated structures to the pixel shader.

As it was mentioned in previous section, the output of the vertex shader consists among others from vectors of texture coordinates and color information for a processed vertex. These vectors are interpolated in perspective correct manner over the surface in order to provide appropriate

value for each of generated surface element. Currently, there are 8 up to 10 of such floating-point four-component vectors.

Besides the interpolated output values from the vertex shader, in the shader model 3.0 the pixel shader has a possibility to obtain other pixel-associated information that was produced by the pipeline: the face register and the position register. Both of these registers are enhancement that allows to access some context information for particular manner. In detail:

- **Face register** contains a scalar boolean value denoting that a processed pixel was generated by rasterization of either front-facing or back-facing triangle. Even thought this value is of boolean nature, the register is declared as a floating-point register: the sign of the value replaces the boolean. The register is accessible only by dynamic flow instructions.
- **Position register** is a two-component floating-point register that contains exact position of processed pixel in screen coordinates. Opposite to the face register, the lack of the position register for lower shader models may be bypassed by utilizing a single texture coordinate for position of particular vertex with aid of texture coordinate interpolation during the rasterization.

Texture is the second most important input source for the pixel shader because it allows an access to rather large set of data stored in on-board memory. Values stored in a texture are read by the **sampler** whose settings including actual texture are influenced by settings of the pipeline. The sample performs sampling according to a sampling filter/mode and a texture coordinate.

Sampling filter depends on the pipeline settings and may be one of these:

- **Nearest-neighbor** (also point sampling) that is used mostly for floating-point textures and non-graphical computation because it allows the shader to retrieve a value that is not corrupted by its neighbors. When sampling, coordinates are rounded down to the nearest lower position on the texel boundary, see below.
- **Linear** that is the most used for high-performance texture mapping and it is widely supported in current hardware. This sampling mode is not supported for full 32-bit floating-point textures but it may be supported for textures of lower accuracy per component, i.e. half. This mode may be denoted as a bilinear because it utilizes two consecutive linear interpolations.
- **Anisotropic** that provides excellent visual results for surfaces that are almost perpendicular to plane of the viewer because the texture is sampled many times according to settings and thus it reduces aliasing issues. This sampling mode is the slowest approach and is intended mostly for texture mapping of textures that contains thin lines or well-defined sharp edges, e.g. text. Similar to previous sampler mode, this mode is not capable to operate on full 32-bit floating-point texture.
- **Mip-mapping** [Wil83] that provides reasonable quality with much less computational afford and requirements. The basic idea is to obtain a value from an image of appropriate resolution in order to avoid aliasing problem. This significantly improves the image quality in comparison to both point and linear sampling mostly for texture down-sampling and has only a little additional performance costs in comparison to the anisotropic sampling.

  Similar to the previous sampling, this mode is intended for texture mapping purposes rather than for general processing. However, it applicable to full 32-bit floating-point textures and allows the shader to access individual resolution levels independently. Besides that the hardware is also capable to perform automatic generation of lower mip-map levels even for floating-point textures thus making this feature useful for averaging of large arrays.

Texture coordinate utilized for addressing of the texture is a floating-point vector with a number of components depending on given texture type. Each of the components is utilized only in a range of $\langle 0;1 \rangle$; out-of-range values are handled according to settings of the sampler, i.e. the sampler may either clamp values to boundaries of the region or take only the fraction part of the pipeline for addressing or provide an application defined value, if overlap is detected.

In contrary to pixels in the render target, texture coordinate points to a **boundary** of the texel rather than its center. This may be little bit confusing when exact texel-to-pixel mapping is required. In such case a texture address has to be offset by a half of a texel size. This approach may also solve problems with inaccurate floating-point arithmetic when performing non-graphic computation with a floating texture even for point-sampling.

Sampler always reads a four-component floating-point vector from a texture no matter how may component were exactly available in a texture. In a case, a texture with 8 or lower bits per texel is utilized the values are stretched into a range for $\langle 0;1 \rangle$ in order to unify the computation on the hardware with saturated arithmetics.

2.2.3.2 Output

Output from a shader is basically a **color** in a form of a RGBA quadruplet that is used as a source for pixel post-processing such as alpha blending. If the render target has depth of 8 bits or less, the only valid values for components of output color are values in range of $\langle 0;1 \rangle$ because it has to be mapped to given bit depth. In order to achieve this goal, the output is saturated to this range for render targets with bit depth of 8 or less.

The floating-point render target has no limitation in its values but it usually have only partial post-processing. For 32-bit floating-point data type an alpha-blending is not supported at all. For 16-bit half the alpha-blending is available only for defined formats of render target, i.e. it has to consist of either single or four-components vectors. Besides that, floating-point render target does not support complete write masks, i.e. the application is not able to block writing for individual color channel, only complete write block is available.

Some of the hardware has a capability to utilize a facility called **multiple render target**. This does not mean anything else then a possibility to output to more then one render target in a single pixel shader execution. The pixel shader is able to store up to four colors in a single execution that may save computation time, e.g. a color map and normal map of the scene and both of these maps are then utilized for advanced illumination techniques [CFX03]. This surely reduces number of generated pixels and rendering calls and thus it reduces the whole computation time.

The drawback of using multiple render targets is the lack of hardware accelerated anti-aliasing, possible restriction in pixel post-processing, and restrictions in data formats of render targets. All render targets have to have same width, height, and bit depth as sum of bit depths of all color components, i.e. it is possible to utilize different format but they all have to fill up area of the same size[12]. For more details, refer to [NVG04].

Besides a color, the shader is able to modify **depth** information associated with a pixel and provide it as an output. This allows depth displacement of a pixel that be useful for a higher LOD when an object or its part is reduced to a 2D image, i.e. billboard [Szi04]. Nevertheless, modification of depth information significantly reduces computational speed because the hardware is capable of optimization by pre-estimation of the depth test before the shader execution.

12 It is possible to mix R32F and A8R8B8G8 because in both of them a pixel requires space of 32 bits.

Such optimization is based on a assumption that pixel, which fails the depth test, is occluded by another one and therefore does not need to be computed in pixel shader at all. Based on that, modification of the depth information in the shader forces the hardware drop precomputed values and execute the pixel shader.

This may seem to be a limitation but it allows simple, yet powerful optimization whenever a scene contains object with complicated and therefore computation time extensive materials. In such case, the scene may build its complete depth information by rendering of a geometry without color evaluation and then use this information in pre-estimation of the depth in order to strip all occluded pixed that has to be computed otherwise. This is basically a slight modification of technique called a **deferred shading** [Las95].

2.2.3.3 Program and Operations

Program of the pixel shader is more limited in comparison to the vertex shader. Usually, while a particular version of the vertex shader has capability of dynamic flow control, the pixel shader either lacks such capability completely or has certain limitation in nesting depth. Also, there is a great reduction in program length for the shader model lower then extension of version 2.0[13]. Regardless of the version, the pixel shader is capable of a simple dynamic control flow by a possibility to discard current pixel.

For shader model 2.0 or lower the total number of available instructions is divided into two groups: arithmetic and texture instruction, i.e. instructions that utilizes samplers in order to obtain a sample from a texture. This limits maximum number of reading from all samplers in a single shader execution.

Otherwise, the number of texture accesses for a single texture is not limited but multiple sampling of a single texture may reduce the performance due to possible texture cache miss and texture memory latency [Hak97]. The worst possible scenario is a situation when a pixel shader is accessing the texture in random manner. In such case, the cache miss is quite high and thus the performance is reduced. The drop-down may be decreased by utilizing multiple samplers for obtaining samples from the same texture.

Similar to that, a **dependent texture lookups** may be a cause for performance drop-down because the hardware has to wait until the shader has evaluated texture coordinates. This situation occurs whenever a sample from a texture is used as a source for coordinate computation of another sample.

Arithmetic accuracy depends in particular shader model. For shader model 3.0 and newer the accuracy is full 32-bit floating-point, for a lower versions of the shader model, the accuracy may be limited, e.g. the hardware can utilize 24-bit accuracy instead of 32-bits[14]. Nevertheless this limitation has almost no influence on quality of synthesized image because at the end the color is quantized to 8-bit depth per component for display. However, it may cause difficulties from general processing on the GPU.

For versions of the shader model lower then version 2.0, the arithmetic my not be even limited in accuracy it may be saturated to range of $\langle -1;1 \rangle$. This is typical for hardware that utilizes fixed-point arithmetic instead of floating-point one due to performance reasons and due to the fact that many of computations for enhanced texture mapping are performable in this range without almost any impact on output visual quality.

13 Also called a version 2.x, where 2.0a is an extension by nVidia and 2.0b is an extension by Ati.
14 24-bit accuracy is typical for shader model 2.0 and Ati-based hardware.

Similar to the vertex shader, the pixel shader is unable to access or modify context of generated pixel during computation. In the case of access, the application has two possibilities: either it can provide such information on its own or it can benefit from enhancement of shader model 3.0, if available. The first possibility utilizes texture coordinates in order to deliver appropriate data to the pixel shader but it requires additional code and data in vertex shader because some of the information is not accessible even in the vertex shader due to locality.

The pixel shader is the last programmable block but not the last customizable. It was intended to perform more sophisticated texture mapping and therefore optimized for high throughput. Thanks to enhancement in both program length and flow control in newer shader model versions, the pixel shader became a programmable block intended to perform various per-pixel operations. Based on its high throughput and a possibility to influence every pixel on the render target the pixel shader is utilized as a computation kernel for general computation running on the GPU.

2.3 Modified Classical Pipeline

Programmable extension of classical pipeline allows implementation of many effects and approaches that either improve the visual quality or increase the performance. Nevertheless, still there are some issues caused by limitations that has to be solved on the CPU thus decreasing the performance of particular solution.

Therefore, there have always been attempts for enhancing the classical pipeline for being able to either process other primitives then a triangle or to provide a new capability. This is an area that has been researched for a longer time then utilization of the GPU for general processing. That is why a large amount of paper for this area exist.

Rather then a complete list of solutions this section contains selected examples of possibilities that may have or already had improved the functionality of programmable classical pipeline. Each example is based on a particular paper and provides a brief description of both problem and its solution proposes in the paper.

2.3.1 General Modifications

2.3.1.1 Multiple GPU: SLI

In order to speedup the rendering, multiple GPUs may also be utilized and with an aid from a hardware device it is possible to make the whole system invisible to the user by solving all issues on both driver and hardware level. An example of such hardware-bases connection is Scalable Link Interface (SLI) bridge utilized by NVIDIA [NVG04, NVT05], which is basically a hardware bridge between two independent GPUs that allows both their synchronization and data sharing.

The goal of this solution is to improve the performance by parallelization of multiple GPUs with only a limited set of restrictions. Currently, the system utilizes two GPUs thus allowing a theoretical speedup of almost two. Nevertheless, in practice the speedup of 1.9 for ideal scene (see below) is expected due to synchronization and data sharing issues. The maximum memory capacity is denoted by minimum of available memory from both connected GPUs, i.e. the link does not create distributed memory system due to requirement of same data for both GPUs.

In order to allow the synchronization, one GPU is denoted as a **master adapter** during system setup. The master adapter is responsible for rendering control and assembling of the output image, if required. The system then is able to run in three modes:

- **Alternate frame rendering** (AFR) mode that threats each GPU as a stand-alone. Every frame is rendered completely on one GPU at once. The speedup achieved in this mode is based on capability of frame buffering available in some graphical interface. The frame buffering allows with a certain limitations of available operations to buffer few successive frame in order to allow the GPU to run independently on the CPU.

  The speedup achievable by the (AFR) is almost double for ideal scenes, i.e. scenes that do not shares temporary data among successive frames and do not forces hard GPU-CPU synchronization, i.e. flushing of the pipeline during back-buffer locking. In order to limit data sharing the application should perform all render to texture operations in a single frame rather than sharing data among frames. Also, it should perform a clear operation for all render targets indicating that no pixels are reused from previous frame.

- **Split frame rendering** (SFR) mode that simply divides the screen to two parts: the upper and the lower part. Each GPU render its own part almost independently and at the end of the frame the master adapter assemblers both parts into a single image digitally. The benefit of such division is the simplified sorting then in a case of other approaches.

  In order to improve the performance, the system has an ability to balance the computation load between GPUs by moving the division line up and down. Nevertheless, the overall performance of this mode is little bit lower in comparison to previous mode due to higher data sharing ration. This mode is assumed for applications that do not utilize frame buffering or limits significantly maximum number of frames for buffering.

- **compatibility** mode that is the last mode. In this mode, the SLI abilities are not utilizes at all: only a single GPU is active. This mode is intended for applications bounded by the CPU and it does not provide any speedup.

The mechanism of SLI is simple and power-full solution that aims on utilizing a brute force in order to improve the performance. Thanks to that, it is applicable for wide range of solutions that requires high-speed rendering of scenes with large amount of generated pixels, i.e. games at most.

2.3.2 Vertex Pipeline Modifications

2.3.2.1 Advanced Displacement Mapping

Vertex pipeline performs primitive preparation in order to allow its proper rasterization. Such primitive is a plain triangle that is stored in and/or transported to GPU accessible memory. The size of such memory is limited and it is usually only a fraction of average main memory size. A majority of the video memory capacity is consumed by textures and other elements utilized in the pixel pipeline whose reduction would lead to decrease of synthesized image visual quality. Therefore, the application has to save by reducing another kind of data: the geometry.

In games as an example of high-performance application, geometry of object is rather simple in order to save both storage requirement and bandwidth of the bus when transporting data from the CPU to the GPU. It is also true that this reduction decreases a number of generated pixels in the pixel pipeline thus saves a computation time. Nevertheless, high reduction has negative effect on

visual quality and in order to prevent degradation of the image an application usually utilizes geometry of various LOD thus increasing the storage requirements.

A possible solution for a conflict of both quality and performance is to compute geometry of appropriate LOD utilizing only a little additional information directly in graphical pipeline automatically. Such solution would then use only crude geometry with low number of primitives thus saving storage requirements and improving visual quality at the same time. Such solution is called a displacement mapping and was already described in Subsection 2.2.2.

One of solutions already available in the hardware is a mechanism of N-patches [Vla01] mentioned in the Section 2.1. It is completely transparent to the user and allows to round the geometry through tessellating of triangular Bézier surface defined by supplied triangular mesh. With a capability of the shader model 3.0 to access a vertex texture a geometry of custom LOD generation is possible.

Nevertheless, tessellation performed by N-patch is **uniform**. The surface is tessellated according to user supplied request and thus it completely ignores complexity of displacement map, i.e. far too many triangles may be generated for almost planar portions while corrugated parts of displacement map suffer from aliasing issues. Also, the tessellation is not view-dependent.

Much more sophisticated is a solution proposed by [Dog00] that aims on elimination supersampling and undersampling issues by adaptivity of a tessellation level. It utilizes recursive division of edges by inserting midpoints according to several tests. Thanks to the fact that new points are midpoints estimation of their position and associated attributes is matter of averaging whose hardware implementation requires only addition and binary shift for division by two.

Normal of displaced vertex is either computer from displacement map directly or extracted from precomputed normal map that the preferred approach. The displacement map then may contain both normals and displacement values thus forming a four component (ARGB) texture whose floating-point version is already accessible in shader model 3.0 even from the vertex pipeline.

The division of edges is an important feature of this approach because it prevents cracks between adjacent triangles, which may have appear if a global criteria of adaptivity would have been used. Also, thanks to the locality of the test the overall implementation could be very simple because it ignores the context of the triangle. Nevertheless, the drawback is that test for division of edges is generally performed multiple times for the same edge thus introducing additional costs.

Let $\boldsymbol{P}_1'$ and $\boldsymbol{P}_2'$ be displaced vertices that defines examined edge. In order to estimate suitability of the division, the algorithm executes four tests:

- **surface normal variance** test that compares new normal to normals of vertices $\boldsymbol{P}_1'$ and $\boldsymbol{P}_2'$. The comparison utilizes a threshold value provided by the application thus allowing customization of LOD generation. This test is sensitive to even small perturbation of the new surface and therefore it decreases the aliasing issue. Nevertheless, it may miss changes in height.
- **local area average height** test that detect changes in height by comparing a difference in summary of heights for an area around newly created vertex and areas around vertices $\boldsymbol{P}_1'$ and $\boldsymbol{P}_2'$ to a threshold value provided by the application. This test is sensitive to changes in height but insensitive to perturbation of the surface thus being capable of handling situating where the surface normal variance test fails and vice-versa.

In order to speedup the process of the estimation, the algorithm utilizes a **summed area table** (SAT) that is a basically a table based bivariate scalar function defined as:

$$\mathrm{SAT}(x, y) = \sum_{j=0}^{j \le y} \sum_{i=0}^{i \le x} D(i, j), \tag{2}$$

where $D(i, j)$ is a displacement height mentioned in Subsection 2.2.2. The SAT then allows a computation of given area just by utilizing additions and subtractions thus simplifies the hardware implementation. It is assumed that SAT computation is invisible to the application.

- **view dependent resampling** test that is restricts the division of the edge according to its size in the screen space. The test utilizes vertices $\boldsymbol{P}_1'$ and $\boldsymbol{P}_2'$ transformed into screen coordinates and estimates its distance that is compared to threshold value provided by the application. Due to the performance reasons, the test utilizes Manhattan metrics, i.e. $d = |x_1 - x_2| + |y_1 - y_2|$. If this test fails no vertex is inserted.
- **refinement limit** test that limits the level of recursive division by a resolution of displacement map. This test stops the recursion by comparing texture coordinates of original vertices and new vertex. Comparison utilizes integers as coordinates resized to a texture resolution rather than comparing original floating-point coordinates. If this test fails no vertex is inserted, i.e. the algorithm refuses to extrapolate displacement values.

After tests are performed for each edge of the triangle, the algorithm divides the triangle into smaller fractions and passes these smaller versions for newer division. If no division appears, the algorithm releases a triangle for next part of the pipeline.

Hardware implementation of the algorithm is an enhancement of classical vertex pipeline and it does restricts a its functionality. The implementation follows a flow graph in the Figure 2.5 that utilizes following blocks:

- **triangle extractor** that obtains a triangle from either a triangle queue or input of the pipeline.
- **new vertex calculation**, which estimates position of new vertex including associated attributes by averaging ends of the edge.
- **vertex displacer** that displaces vertices according to displacement map. All vertices are displaced only once, i.e. usually only new midpoints are displaced.
- **transformation stage** that is simply performs all computations required in order to transform vertex to screen space, i.e. it performs clipping and projective/viewport transformation. The back-face culling cannot be performed in this stage because the back-facing triangle in low-res model may contain front-facing triangle due to displacement mapping.
- **tessellation tester** that runs tests mentioned above and decides whether the triangle is going to be divided or not.
- **tessellator** that divides triangle according to results of both tessellation tester and vertex displacer.
- **triangle queue** that is a buffer for triangles generated by tessellator that may be utilized for another recursive division.

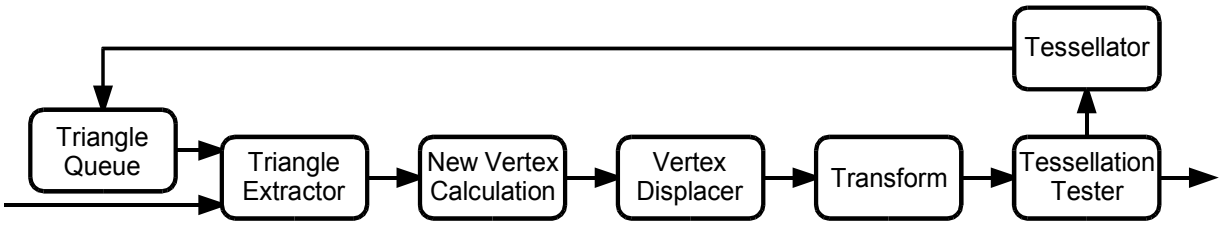


Figure 2.3: Modified graphic pipeline for adaptive displacement mapping. This flow-graph basically replaces both High-order Primitive Tessellation and Transform&Lighting as shown in Figure 2.1.

The solution utilizes as many parts of original common vertex pipeline as possible and it also utilizes only simple operation such as addition, subtraction, and binary shifts. The flow graph in Figure 2.5 also does not disturb pipeline behavior of the vertex pipeline. The possible drawback of this solution may be the division of transformation and lighting operations into two both separate and distant blocks while the GPU performs both these operations in a single vertex shader.

However, this drawback may be eliminated by dividing the vertex shader code into part that handles transformation only and the rest. The division may be performed automatically in driver by eliminated unused instructions from the shader code. This makes the mechanism completely invisible to the application while the shader pipeline still retains it programmability.

2.3.2.2 Clipping in Homogeneous Coordinates

Clipping is an internal part of the vertex pipeline. It process output of primitive competition part and its output is passed to triangle setup for rasterization. Due to both performance and the fact that it is natural part of image synthesis process, it is hard-wired. It seems that this solution shall offer the best performance possible but with increasing number of primitives per scene even this part may be the bottleneck.

According to observations [Ska04], algorithm utilized by current hardware is probably a version of Sutherland-Hodgman. This algorithm works in non-projective space thus requiring a homogeneous divide before the clipping is performed. Also, during the computation division may be required. The advantage is a possibility to utilize similar group of operation for all four sides just by switching coordinates. Nevertheless it still requires wide range of operation to be implemented in that group.

Solution proposed in [Ska04] may a possible approach for this problem even thought this solution utilizes P^2 space, i.e. projective enhancement of 2D space. It performs a clipping against rectangle. It benefits from features of projective enhancement: let consider $\boldsymbol{p_1} = [x_1, y_1, w_1]$ and $\boldsymbol{p_2} = [x_2, y_2, w_2]$ be two points in P^2. Let a $\boldsymbol{l} = [a, b, c]$ by a line defined as:

$$ax + by + cw = 0,\ w \neq 0\ . \tag{3}$$

For such case, a line *l* defined as crossproduct:

$$\boldsymbol{l} = \boldsymbol{p_1} \times \boldsymbol{p_2}\ . \tag{4}$$

Besides that the proposed solution benefits from a principle of duality. In a P^2 space a point may a considered as a line in its duality. Therefore a line $\boldsymbol{l}$ that is defined by two point $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ is basically a result of intersection of lines $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ as it is defined in the Equation (4).

In order to estimate the side, which is intersected by the line, the algorithm utilizes a series of test benefiting from a line definition in the Equation (3) that basically divides a plane into two halves. By substitution of clipping rectangle corner points in the equation a vector of boolean flags is

created according to the left side of the Equation (3), i.e. the flag is true if the left side is less or equal to zero.

Then by utilizing a table to all possible situation, an intersected border is determined. Afterwards, the algorithm performs line intersection based on features of P^2 space mentioned above. The algorithm is visualized in the Figure 2.4.

The important feature of this algorithm is a possibility of parallelization of border for intersection. Also, the algorithm utilizes only table lookups and crossproducts and comparisons are done against zero thus according to [IEE85] this operation is reduced to a single bit test. Operations are performed in projective space thus removing a requirement for homogeneous divide and improving the accuracy thanks to lack of division.

All features mentioned above simplify hardware implementation of the algorithm thus probably improving the performance. The only drawback of the algorithm is the fact that is it strictly bound to 2D space and its projective enhancement: the 3D extension is currently in research.

$\boldsymbol{l} \leftarrow \boldsymbol{x}_A \times \boldsymbol{x}_B$
$\boldsymbol{c} \leftarrow [0, 0, 0, 0]$
for each k in $\{1, 2, 3, 4\}$:
    if $\boldsymbol{l} \bullet \boldsymbol{x}_k \geq 0$:
        $c_k \leftarrow 1$
    else:
        $c_k \leftarrow 0$
if $\boldsymbol{c} \neq [0, 0, 0, 0]$ and $\boldsymbol{c} \neq [1, 1, 1, 1]$:
    $\boldsymbol{e}_A \leftarrow \text{table}_A[\boldsymbol{c}]$ $\boldsymbol{e}_B \leftarrow \text{table}_B[\boldsymbol{c}]$
    $\boldsymbol{x}_A \leftarrow \boldsymbol{l} \times \boldsymbol{e}_A$ $\boldsymbol{x}_B \leftarrow \boldsymbol{l} \times \boldsymbol{e}_B$

Figure 2.4: Clipping algorithm of line defined by points $\boldsymbol{x}_A$ and $\boldsymbol{x}_B$. Line is clipped by rectangle $[\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4]$. Algorithm utilizes an edge table table_A and table_B. For details, refer to [Ska04].

2.3.2.3 Visibility Solving in Vertex Pipeline

The fill rate is the major bottleneck in current graphic hardware mostly experienced in game application. It is caused by a fact that while the large number of pixels are processed but only a small fraction stays visible due to occlusion. Even thought this bottleneck is caused by the pixel pipeline, its reduction may be achieved before the geometry leaves the vertex pipeline, as it is proposed in [Ail03].

Surely, major performance improvements may by achieved by simply presorting rendered elements according to their depths. However, application such as games usually tends to sort rendered object in order to minimize state changes, rather than the visibility because frequent state change is an expensive.

In order to decrease the fill rate, a **delayed occlusion test** is introduced. This test decrease the fill-rate by performing an occlusion test on per-triangles basis. Incoming triangle is tested against already known occlusion information and if it is still visible then it is used to update the occlusion of the scene and stored into a **delay stream** for later testing. When a flush request is detected, processing of triangles stored in delay stream starts. In this processing a triangle is tested once more against collected occlusion information in order to decide its visibility. Visible triangles are then passed to the pixel pipeline.

Besides geometry information, the delay stream contains render state changes so the pipeline is able to render scene appropriately. Storing vertices into a stream may include a simple

compression based on a history[15] as described in [Ail03]. It is sure, this stream has rather limited in size and certainly it is not possible to store complete scene. Therefore, the stream is processed not even when flush is issued but also if it is filled to a given percentage of its full capacity. The rest of the capacity is utilized for load balancing.

Delayed visibility testing emulates simplified version of scene depth estimation, i.e. differed shading [Las95]. Theoretically, delayed testing may provide speedup of two up to four in comparison to scenes rendered without delayed visibility testing. Important aspect of this solution is the fact that it does not modify the order in which triangles were sent to the pipeline by the application. This make this solution transparent to an application because it does not introduces any limitation to incoming stream of geometry.

Nevertheless, this method is unable to decide the occlusion whenever per-pixel rejection by either stencil or alpha test is enabled or whenever a pixel shader modifies depth information associated with a pixel because decision of possible visibility is executed in the pixel pipeline. In such case, triangles with these properties are passed through without any occlusion tests.

2.3.3 Pixel Pipeline Modifications

2.3.3.1 Quad as Rendering Primitive

Another possibility how further extend the pipeline is to allows different and more complicated primitives to be rendered with support of hardware as natural as triangles. One of such primitives descried in [Hor04] is a **quad**, i.e. a polygon of four vertices. The reason for this enhancement is a fact that every modern polygonal modeler gives best results when a quad is used as the basic primitive because it produces pleasant and reasonable results when a mesh smoothing is applied than a triangle.

However, current hardware has no support for rendering a quad naturally: the quad has to be divided into a two or more triangles. The disadvantage of the division is a fact that for a simple diagonal split a discontinuities in attributes such a texture coordinates appears either along the diagonal or across horizontal span in two-fold linear interpolation performed in scan-line rasterizer. A possible reduction of such discontinuities is to refine the quad into a matrix of $n \times n$ smaller quads where discontinuities are not longer visible. Nevertheless, such solution produces a large number of primitives which may increase the load on the GPU.

The most important problem that has to be solved is an interpolation of both depth information and attributes such as texture coordinates in projective expansion of Eucleudian space. For such purposes a modification of Barycentric coordinates is utilized. The modification is denoted as a mean values and it is based upon defining a set of coordinates $\mu_i(\boldsymbol{v})$ of given point $\boldsymbol{v}$ with respect to a quad $[\boldsymbol{v_0}, \ldots, \boldsymbol{v_3}]$, where $\boldsymbol{v_i} = (x_i/w_i, y_i/w_i)$ is a projected vertex in screen coordinates. Such coordinate has to satisfy:

$$\sum_{i=0}^{3} \mu_i(\boldsymbol{v})(\boldsymbol{v_i} - \boldsymbol{v}) = 0 \,. \tag{5}$$

Barycentric coordinates $\lambda_i(\boldsymbol{v})$ are then determined as a normalization of coordinates $\mu_i(\boldsymbol{v})$:

$$\lambda_i(\boldsymbol{v}) = \mu_i(\boldsymbol{v}) / \sum_{j=0}^{3} \mu_j(\boldsymbol{v}) \,, \tag{6}$$

$$\mu_i(\boldsymbol{v}) = \frac{1}{r_i(\boldsymbol{v})} \left(\tan\left(\frac{1}{2}\alpha_{i-1}(\boldsymbol{v})\right) + \tan\left(\frac{1}{2}\alpha_i(\boldsymbol{v})\right) \right) , \tag{7}$$

15 The history compression is based on removing repeating vertices and replacing them by an index.

where $\alpha_i(\boldsymbol{v})$ is a signed angle at point $\boldsymbol{v}$ in triangle $[\boldsymbol{v}, \boldsymbol{v_i}, \boldsymbol{v_{i+1}}]$ and $r_i(\boldsymbol{v})=|\boldsymbol{v}-\boldsymbol{v_i}|$, i.e. a distance between $\boldsymbol{v}$ and i-th vertex. The important characteristics of $\mu_i(\boldsymbol{v})$ is that they has has identical signs when the point is inside the quad even if the quad is self-intersecting.

In order to be able to compute attribute for each point of projected quad, the algorithm utilizes a variable $\lambda'_i(\boldsymbol{v})$ to compute exact location in projective space. Basically, it is simple perspective enhancement of Barycentric coordinates. Appropriate attributes are computed by analogy to $\lambda'_i(\boldsymbol{v})$. However, it is possible to interpolate attribute a_i/w_i and divide it by interpolated $1/w_i$ value. Projection enhancement of Barycentric coordinated is defined as:

$$\lambda'_i(\boldsymbol{v})=\frac{\lambda_i(\boldsymbol{v})}{w_i}/\sum_{j=0}^{3}\frac{\lambda_j(\boldsymbol{v})}{w_j}=\frac{\mu_i(\boldsymbol{v})}{w_i}/\sum_{j=0}^{3}\frac{\mu_j(\boldsymbol{v})}{w_j}. \tag{8}$$

The proposed implementation utilizes a scan-line renderer and benefits from a knowledge of maximum number of spans per a single line, which is one for triangles and two for quads. This simplifies organization of structures required for list of active edges. Besides that the implementation may also benefit from a fact that Equation (7) can be expressed with help of following:

$$\tan\left(\frac{1}{2}\alpha_i\right)=\frac{r_i r_{i+1}-(\boldsymbol{s_i}^{\mathrm{T}}\cdot\boldsymbol{s_{i+1}})}{|\boldsymbol{s_i}\times\boldsymbol{s_{i+1}}|}, \tag{9}$$

where $\boldsymbol{s_i}=\boldsymbol{v_i}-\boldsymbol{v}$. This allows computation of coordinate $\lambda_i(\boldsymbol{v})$ from projected point $\boldsymbol{v}$ in screen coordinates. Based on a fact that dot product is linear and both cross product and r_i^2 are quadratical, the implementation may utilize incremental modification from the pixel to the pixel according to a initial values generated in quad setup phase, similar to triangle.

The benefit of this solution is based on modification of rasterizer only, the rest of the pipeline remains the same. Even though the pure hardware solution is not available yet, results were generated utilizing the hybrid GPU solution aided by the CPU for geometry setup. The hybrid solution benefits from a possibility to determine whether the projected point is inside or outside of rendered quad. For more details and exact description of algorithm used, refer to [Hor04].

2.3.3.2 Displaced Pixels

Besides introducing a new rendering primitive that surely improves the visual quality of the rendering with only a minimal performance effect an expectable extension to the pixel pipeline is to enhance output of the pixel shader towards a **context modification**, i.e. modification of position. Currently, shader model allows depth modifications but more effects may be achieved when a depth would be enhanced by position modification as it is proposed in [Die04].

If implemented, this modification allows implementation of algorithms based on accumulation of values in multiple accumulators according to a definition such as histogram computation or other solution from image analysis area. These kind of algorithms are currently implementable on graphic hardware with only high number of passes thus reducing the performance even under the CPU level.

Another area which may benefit from relocatable pixels are particle systems. This enhancement would allow a particle system to be both simulated and even rendered in a single pass without additional computation and storage costs by utilizing a multiple render targets. Currently, thanks to vertex textures introduced in shader model 3.0, particle systems do not require GPU→CPU data traffic and may be rendered in two-pass algorithm.

Similar to particle systems this pixel processing enhancement would allow an effective direct rendering of curves by simply applying an appropriate definition of the curve. Also, it is assumed that this extension would allow an effective rendering of a fur with fur parameters stored in textures and based information the pixel may be reallocated to a proper position. Thanks to that, a effecting the LOD could be achieved just by increasing a number of rasterized pixels whenever a furry surface gets closer to the camera.

Nevertheless, the extension proposed in [Die04] does not offer a solution for several implementation issues. It is clearly visible that such implementation would require both color and depth buffer with random access, which may reduce the reflectivity of memory write operation even if caching based on coherency of changes is enabled.

Also a requirement of additional clipping operation after the pixel has been reallocated in order to avoid invalid memory access reduces the performance even thought such clipping is similar to a guard-band mechanism described in previous sections. However, this is not the major problem of the implementation.

The major difficulty of the implementation is disturbance of pipeline structures that is a fundamental for current GPUs. There are synchronization issues in the pixel post-processing where a coherence in operations such as an alpha-blending or a stencil/depth-test is required. In a current graphic hardware the synchronization is probably solved on a primitive level thus allowing pixel pipeline to run almost unsynchronized, which surely improves the overall throughput of the pixel pipeline.

2.3.3.3 Order-independent Alpha Blending

Another possibility for improvement if the pixel pipeline is to solve the problem of an alpha-blending × depth buffer, i.e. to provide order-independent alpha-blending. Whenever an alpha blended surfaces occur in a scene the application has to handle them separately because the depth-buffer visibility does not consider the transparency and thus it handles transparent pixels as they were a solid.

The solution proposed by [Ail03] utilizes a FIFO stream in order to store geometry of triangles and coordinates of their unprocessed pixel. For optimization reasons, pixels are groups into 8×8 pixel block that each has a mask of unprocessed pixels. The solution requires a division of pixel shader program into a alpha-modifying part and color/depth-modifying part. This can be performed automatically and therefore invisible to the application as a part of a device driver.

Every time a triangle with enabled order-independent transparency arrives, it is rasterized, and alpha-modifying shader code is applied. Obtained data are then stored into the stream for later processing as it is described by the Figure 2.3 below.

```
mark hidden pixels as processed
while stream.Count > 0:
        remove processed triangles from stream
        clear depth buffer
        walk the stream and find highest depth value for each pixel
        walk the stream:
                if depth value is equal to found one:
                        process pixel
                        blend pixel with output
                        mark pixel as processed
```

Figure 2.5: Order-independent alpha blending pixel processing algorithm.

2.3.3.4 Simple and Effective Anti-aliasing

Another performance issue in the pixel shader pipeline is a question of proper **anti-aliasing**. Currently, the hardware may perform either full screen anti-aliasing that is rather expensive or anti-aliasing of edges. In latter case, the anti-aliasing is performed even on edges that are not part of the silhouette. In majority of cases these are the only ones where the alias is visible at most and therefore their proper estimation may save a computational power as proposed in [Ail03].

The solution utilizes a simple set of flags for each triangle and its edges. This set indicates whether is it a inner edge or discontinuity edge. When a triangle arrives, a lookup into a edge hash table is made: if edge is found then the record is remove from a hash table and newly incoming edge is marked as closing one. Otherwise, a new record is inserted into the table and the edge is marked as opening one.

During the rasterization, each pixel has a counter assigned. This counter increments whenever an opening edge crosses it and decreased in the case of closing edge. At the end, if the counter is zero then this pixel is not an issue for anti-aliasing because it is either inner pixel or resides a shared edge. For the rest of pixels a supersampling is applied. This approach shall reduce number of anti-aliased edges to 3—20 % of the total number but it increases memory bandwidth at the same time.

3 GPU Languages and Programming

The GPU provides great amount of functionality accessible by the application in order to either compute general purpose solutions or build visual effects. Nevertheless, until now GPU limitations and capabilities were only described in previous chapters, there was no comment on an actual programming language and/or tool. Therefore, in this chapter an overview of selected available languages is presented.

For each language, its features are described in order to familiarize the reader with languages that are utilized by solutions described in this document. Only major features are mentioned, for details, please refer to particular language specification. Languages are divided according to an intention of their utilization even thought this intention may not force the utilization for such purposes.

3.1 Graphical Purposes

Languages intended for graphical purposes are the most widespread languages for the GPU programing on common graphic hardware. At their beginnings they were intended for enhancing of a visual output during image synthesis, i.e. allowing to program a part of the pipeline. Therefore they have same attributes as the GPU itself.

In a majority of cases their construction is based on shaders that are utilized for various off-line rendering system, such as RenderMan shading language [Han90]. Opposite of this system, the important feature is a division of shader code into two parts: the vertex and the pixel shader code for the sake of the performance. Such division accommodates better to the underlaying structure of the GPU that is basically a pipeline with two processors.

According to [Mar03] this feature becomes an important when branching comes to play. In the case where the compiler does the division between underlaying processing units, the resulting code may be far too ineffective due to branching in cooperation with inappropriate mixing of instructions for the first and the second processing unit. On the other hand, the division made by the application forces the developer to optimize the code with the division in mind thus resulting to a code of higher performance. Also, it keeps the programming model simply enough.

Languages for graphical purposes may be divided into two groups according to their complexity:

- **low-level languages**, i.e. basically assemblers. These languages are primary languages for the GPU and they were utilized since the beginning of modern GPUs. They exhibit SIMD behavior in their instruction as it was described in the Section 2.2 and they are unable to bypass even minor limitations of the GPU. Currently, they are not utilized for both graphical and general computing on GPUs due to complexity of codes written with them. Therefore, assemblers for the GPU are not described further in this section at all.

- **high-level languages** that are utilized instead of assembler languages in a majority of currently available solutions. In the most cases they are based on C-language with certain restriction of its capabilities based on features of the GPU. They also allow handling of few minor yet annoying limitations such as lacks flow control and non-SIMD behavior automatically by the compiler thus simplifying the process of implementation. Currently, there are many of these languages and those mentioned below are utilized at the most.

3.1.1 Direct3D: HLSL

High Level Shading Language is a GPU language introduced in a version 9.0 of DirectX [DX05]. It shares both syntax and semantics with the Cg language [Mar03] thus making it compatible on the source code level. In a limited manner the HLSL may be considered as a subset of the Cg even thought this it mentioned in neither the HLSL documentation nor the Cg specification.

In its fundamentals, the HLSL is based on syntax of C language with a few features from C++ such as boolean data type, structure definition, and comments. Due to limitations of the GPU, it is not able to implement complete subset of C language features, i.e. the HLSL is not able both to operate with pointers and to perform bit-wise operations. The is caused by incapability of the GPU to operate with independent bits of data types and the nature of shader accessible read-write memory. For more details on GPU limitations refer to Section 2.2.

3.1.1.1 Data types

Besides that the HLSL supports C-like preprocessor thus allowing a conditional compilation. It also extends the C languages by introducing features that allows to utilize capabilities of the GPU. It is mostly matter of new data types including operation definition. The available set of data types is striped of pointers and majority of integers with exception of the `int` data type and it is enhanced by a boolean data type and a new floating-point types.

The boolean is utilized for if-branching both static and dynamic. In the case of floating point numbers the set is enlarged by types of lower accuracy then the `float` [IEE85] such as `fixed` that represents fixed-point real number and `half`, which is a floating-point number of half precision. These types are utilized due to either restrictions of older GPUs or performance, see Section 2.2

In order to bypass differences of various shader models the compiler my utilize type with higher accuracy then the type requested, i.e. the `fixed` type may be replaced by the `half` type during compilation due to lack of support for the `fixed`. Similar to that, the `int` data type is replaced by the `float` for the majority of cases with exception of loop iterating or array addressing because the GPU usually lacks read-write temporary integer registers.

As it was already mentioned in Section 2.2, the GPU supports operations with vectors of up to four-components in length. It would have seen that this can be implemented with the aid of C arrays. However, in such case the compiler would not be able to distinguish between real array of four floats and a vector of four float components [Mar03]. Therefore, another enhancement the the data type set is introduces: the vector.

The **vector** may by of any both numerical and booleans data type and its length is restricted to four. Syntactically, the vector is denoted as `float`*n* where *n* is a number of vector components. Similar to vector operation supported by the GPU, the HLSL allows masking and swizzling of vector components, i.e. it allows to select vector components, modify their position, or multiply a single component in order to form a new vector of possible different length.

Similar to the vector, the HLSL support a **matrix** data type of size up to 4×4. This type is emulated by a set of vectors in column-major order. The emulations allows simple yet fast implementation of matrix-matrix and matrix-vector operations, e.g. matrix-matrix multiplication may by implemented as a set of dot products. Syntactically, the matrix is specified similar to the vector and sizes are specified in row-column order, i.e. matrix of 3×4 is denoted as `float3x4`.

It would be good to note, that all binary numerical operators such as addition, multiplication, or division operate over **individual elements** by default. If matrix operation such as matrix-vector multiplication is required a special function has to be utilized rather than multiplication operator.

The HLSL allows addressing of individual elements via either dot operator or array approach. For example, let there be a need to address a element in second row and first column of matrix `m` then such element is selected as `m._21`, `m._m10`, or `m[1][0]`. With exception of the array approach the HLSL allows vector-like operation with elements such as swizzling and masking in order to create new vectors and/or matrices.

Also, the HLSL supports **arrays** and **structures**. Both are accessible as it is common in C language, i.e. via either square bracket operator or dot operator. Syntactically, they are almost the same as in C with an exception of structures, which utilizes a C++ style of definition.

3.1.1.2 Flow Control, Operations and Functions

Flow control in the HLSL is supported according to particular shader model and the compile is able to bypass few of limitations of the underlaying GPU. The HLSL supports wide range of control flow constructions with exception of non-structured means of flow control such as label jumps, i.e. `goto`.

Nevertheless, the compiler is able to bypass minor limitations of a particular shader model: it is possible to compile code that contains dynamic branching by computing both branches and utilizing an instruction that selects from two values according to the third one. Similar to that for a static loop the compiler may expand the loop to satisfy requirements of the particular GPU.

Calls of user defined functions are also compiled according to actual shader model. The compiler is able to bypass a lack of real subroutine call instruction by inserting a function call directly thus increasing the shader code length. Nevertheless, the compiler is unable to bypass a prohibition of both direct and indirect recursion. Parameters a handled according to their modifiers (see below) and unused temporary registers are utilized to create their local copies if required.

In any way, the compiler is unable to bypass a limitation in length of the code, i.e. number of instructions. This may be crucial when a loop expanding together with low pixel shader version is used thus resulting into compiler error. Then the only solution for the developer is either to utilize multiple rendering passes or to follow a proposed automatic decomposition of code described in Subsection 3.3.1.

In order to be able to compute basic vector operation, the HLSL utilizes a set of **built-in functions**. These functions are usually a direct counterpart of particular instructions. The complete set can be seen in [DX05] but actual set recognized by the compiler depends on particular shader model. If an operation is not available such as reciprocal square root utilized in vector normalization the compiler does not attempt to emulate the functionality via textures even thought it is possible.

All operations with exception of branching constructions exhibit a SIMD-like behavior even thought the underlaying instruction is capable to process a single scalar at the time. This means

the compiler is able to emulate the SIMD behavior by increasing the number of generated instruction per operation.

Common mathematical functions implemented in a form of a binary operator such as addition and multiply operates over individual elements of utilized vectors and/or matrices. If a vector-matrix multiplication is required a special function `mul` has to be used. Similar to that, vector operations of dot product and crossproduct are implemented in a form of a `dot` and `cross` instruction rather than an operator.

The HLSL supports **user defined functions** with a syntax similar to the C language. Each shader program should contain at least one function that serves as a shader entry point. Functions may be overloaded similar to C++, i.e. the function may have similar name but they have to differ in number and/or type of parameters.

The only enhancement besides the semantics (see below) is a possibility of parameter modifiers that precedes the parameter data type in its syntax. The modifier influences actual storage and behavior of particular parameter. Supported modifiers are:

- `uniform` that denotes a constant accessible from the application.
- `in` that denotes an input parameter. This is default behavior.
- `out` that denotes an output parameter. This is usually utilized for shader output values.
- `inout` that denotes an input and output parameter. This modifier saves a temporary registers that are utilized for parameters similar to a stack in the case of the CPU whenever a function call is requested.

3.1.1.3 Semantics

Besides modifications of the data type set, the HLSL introduces new construction, which denotes a semantics of particular variable and/or function parameter. This construction allows a shader code to connect to particular registers of the GPU thus allowing to access and modify data processed by the pipeline.

It also allows to connect a variable to a particular constants register that can be set by the application directly bypassing the interface of the shader. This also allows a backward compatibility with applications that utilize shader codes in assembly language because in such case the application has to reference constants by the register name instead of their logical name in the shader code.

Semantics is utilized mostly in the entry point of the shader in order to access data from the pipeline. It may by also utilized for a return value of the function to denote the output of the shader code to the pipeline. Syntactically, the semantics is inscribed after the identifier of particular code element, to whom is the semantics applied, e.g. a position of the vertex that is going to be processed by the vertex shader code is denoted as `in float4 pos:POSITION` in parameters of the entry point function.

3.1.1.4 Shader Data Input and Output

Input of the shader utilizes two possible sources of pipeline data: varying and uniform. The term **varying** denoted a set of registers that contains data provided by the pipeline, i.e. processed elements. The HLSL obtains these data as parameters of entry point. These parameters has to have both `in` modifier and appropriate semantics defined.

Uniform denotes basically set of constants including textures provided and accessed by the application. These constants may be passed to the code in a form of either a parameter of entry point function or a global variable. In both cases a modifier `uniform` has to be specified. Semantics is optional and can be utilized for connecting the constant to the constant register.

Output of the shader allows the code to deliver processed element for further processing to the pipeline. The HLSL specifies an output as either `out` parameter of entry point or return value of the entry point function or combination of both. In either cases, semantics has to be specified and compiler performs a compile-time checks in order to prevent a shader from ignoring a obligatory outputs, e.g. the vertex shader has to set a value of a parameter denoted by `POSITION` semantics.

3.1.1.5 Effect Framework

In order to simplify the use of the shader in the application the HLSL supports concept of effects. An effect is a compound set of **techniques** where each technique has **passes** specified that are required for proper rendering of effect. For an each pass pipeline settings are provided including shader code and shader entry points, if the shader is utilized. The whole effect is usually stored in a well-arranged plain text FX file.

From within the application code, the effects loading and constant access is matter of simple function calls. Application of the effect the consist of `begin-end` pairs for both effect and pass calls and a geometry rendered inside these pairs. Even thought this may seems to be a tool aimed only on composition of visual effects the FX framework may be utilized even for general processing purposes for which a pass in the FX technique is equal to a step in the algorithm.

3.1.2 Cg

The Cg language [Fer03, Mar03] is another language based on C language. Is is very close to the HLSL and according to its features the Cg can be denoted as a superset to the HLSL. This makes the Cg and the HLSL compatible on a source code level. Thus, this subsection contains only description of difference between both languages rather than enumeration of all features that were mentioned in previous subsection.

The major feature of the Cg is a platform independence, i.e. it does not distinguishes between graphic application programming interface (API). It is possible to utilize it together with either OpenGL[16] or Direct3D via a special layer that converts Cg function calls into appropriate calls of underlaying API.

The Cg introduces a sophisticated use of **profiles**. The profile is a definition of capabilities and restriction of particular shader model, pipeline part, and graphic API. As opposed to the HLSL that utilizes profiles only for check for limitations during a compilation, the Cg allows an overloading of functions according to specified profile.

This feature is important mostly for generating visual effect where the developer may decrease the quality of the effect of a sake of ability to run on older hardware. For the general processing this feature is not utilized because the developer usually focuses only on a particular hardware combination due to the fact that older hardware may have far too tight restrictions.

Another interesting feature available in the Cg language is a possibility to add functions to a structure thus making it a limited version of classes known from the C++ language. The Cg also

16 In the case of OpenGL the Cg may be able to run better on nVidia based hardware due to an ability to utilize various proprietary extensions developed by the nVidia thus making itself more efficient for this hardware.

support a mechanism of interfaces that is utilized for defining of general functionality available in structures. Such features of structures allow the application to define an array of distinguishable elements implementing same interface that is utilize for uniform handling of these structures in the shader code [Mar03].

The Cg is a language utilized in many application. Its first version were available prior to the HLSL and the GLSL[17]. Similar to the HLSL it is able to utilize different hardware and it is able to bypass minor limitations. As distinct to the HLSL, it is capable to cooperate with both DirectX and OpenGL even thought a compatibility issues may appear on some combination of OpenGL and graphic hardware due to driver instability. For a complete specification as well as list of examples, refer to [Fer03].

3.1.3 OpenGL: GLSL

Another language that is tightly connected to a specific platform is the OpenGL Shading Language (GLSL) [Kes04]. This language is a part OpenGL 2.0 specification [Seg04] and allows a high-level programming of the shader utilizing pure OpenGL. Similar to previously mentioned languages, the GLSL is based on C-languages and shares many features of the HLSL/Cg.

This subsection contains an overview of differences to the HLSL rather than a complete description of the language. The difference of the GLSL to the HLSL is a different manner of pipeline data access philosophy, different syntax of added data types, slightly different meaning of some mathematical operators, lack of profiles, and finally lack of effect (FX) mechanism without any replacement.

As distinct from the HLSL and/or Cg, the GLSL lacks a mechanism of **profiles**. The GLSL expects a minimal functionality as it is specified by a particular OpenGL Extensions [GLE04] required[18] for compilation and application of the shader code. The drawback of this approach is based on a fact that OpenGL standard specifies only the functionality of valid actions rather than a complete specification thus allowing hardware vendors to handle invalid situation on their own.

The actual implementation of extensions is provided by hardware vendors, which may allow the GLSL even for hardware that does not provide all expected functionality such as numerical accuracy. This may lead to both instability of the solution based on the GLSL or appearing of artifacts in the case of visual effects. The solution for that is to test for availability of extension together with capability check of utilize hardware via its identification string[19].

3.1.3.1 Data Types

Data types utilized by the GLSL are pretty the same as for the HLSL with only a few differences mostly in syntax area. Similar to the HLSL, integers are supported but they are expected to be of 16-bit accuracy. It also assumes that integers are processed naturally without hidden conversion to floating-point numbers as it is common among HLSL/Cg compilers. Floating-point are supported without any further distinction between various accuracies because the GLSL expects a particular functionality rather than providing a backward compatibility with older hardware.

Vectors are supported with a slightly different syntax, i.e. a vector is denoted as `tvecn` where `n` is number of components and `t` denotes the type: `i` for integers, `b` for boolean, empty for

17 OpenGL Shading Language, see Subsection 3.1.3.

18 In order to utilize the GLSL the application has to be able to access `ARB_shader_object` and either `ARB_vertex_shader` or `ARB_fragment_shader` or both.

19 The identification string can be retrieved via function `glGetString`.

floating-point. Components of vectors are accessible via dot operator similar to the HLSL. Swizzling and masking is supported too.

Matrices are supported as well and similar to vectors, the syntax is different then that of the HLSL, i.e. `t`mat`n` denotes a matrix with elements of type *t* and size of $n \times n$. All matrices are stored in a column major order. Access to individual components of the matrix is available via array-like approach and their swizzling and/or masking is not supported.

3.1.3.2 Flow Control, Operations and Functions

Flow control support in the GLSL is similar to that of the HLSL. The language supports wide range of constructions such as loops and if-branching with exception of non-structural jumps such as `goto`. However, the actual compilation results depend on implementation provided by the hardware vendor.

As it is common among GPU languages, all operations exhibits SIMD-like behavior. Mathematical binary operator of addition and multiplication are component-wise operations in a majority of cases. Nevertheless, for both vector-matrix and matrix-matrix **multiplication** the operation performed is similar to appropriate **mathematical operation**, i.e. vector-matrix multiplication applies a matrix to a vector. Component-wise version of these operation is available in a form of a special function.

Similar to the HLSL, the GLSL provides a comparable set of built-in functions. Also, it allows definition of user functions. The syntax is similar to both the C language and the HLSL. Parameter modifies are supported with an exception of `uniform` that is omitted due to a different approach for pipeline data access. Overloading is supported similar to the C++, i.e. the function may have similar name but it has to differ in either number of parameters or their types.

3.1.3.3 Shader Data Input and Output

As opposed to the HLSL and the Cg, the GLSL does not support semantics. Rather then that it utilizes a system of predefined variables and constants that is close to a mechanism available in RenderMan shaders [Han90]. All predefined elements are denoted by prefix of `gl_` and the GLSL distinguished following forms:

- **Predefined variables** that are utilized in order to pass data required by the pipeline from the shader.
- **Varying variables** that allow a data communication between the vertex shader and the pixel shader. The GLSL provides a set of predefined variables that represent common data passed to the pixel pipeline such as texture coordinates, diffuse color, etc. This set is available in both vertex and pixel shader.

  The developer is able to define own varying variables in order to transfer data between custom vertex and pixel shader. The connection between variables is established during linking phase via compliance of identifiers. The varying variables are denoted by modifier `varying`.
- **Uniform variables** that contains either data set by the application or settings of the pipeline. This allows a shader to access various data from pipeline settings such as transformation matrices, lights, etc. Uniform variables are denoted by modifier `uniform`.
- **Attributes** that contains vertex data passed to the shader by the application.
- **Constants** that provides various maximum values for pipeline settings, e.g. maximum number of lights, maximum number of temporary variables, etc.

3.1.4 Sh

Even thought mentioned programming languages for the GPU are high level, the mechanism of their connection to the underlaying graphical API is a rather low-level. If a mechanism such as effect framework is not utilized, the developer has to perform a list of API calls in order to utilize a single shader program.

Also, the shader code is not a part of the application source code processed by the compiler, it either resides in an external text file or is stored in a string inside. Nevertheless, its compilation including syntactical and semantical check is performed either at runtime or by separate tool off-line thus complicating the development.

A library that tries to address this annoyance is the Sh [McC02, McC04]. It is based on a mechanism of metaprogramming that is basically a possibility to insert code written in different language into the source code and provide binding by default, i.e. it automatically provides code necessary for utilizing given expression in different language such as the SQL query.

The idea of metaprogramming utilized in the Sh is based on a fact that a proper sequence of function calls from a specific API may lead to a construction of code in different language. If this transcription of function calls is done during a setup phase the performance of such solution is almost the same as it would when compiled all at once.

Also, this idea of function calls allows implementation of such system without requirement of special purpose compiler. When underlaying language is capable of operator overloading and other similar language constructions the inserted code does not differ from original code in a different language. The Sh utilizes the C++ language together with the OpenGL in order to create a tool that allows definition of a shader inside the application code.

Thanks to the level of abstraction, the Sh can be considered as a hybrid between languages closer to the hardware and languages with higher level of abstraction, see Section 3.2. The shader in the Sh library is then a class or a template with fields that contains both shader code and application accessible variables, i.e. uniform variables.

In order to obtain data from the pipeline and provide processed data back channels are utilized. A **channel** is basically a sequence of elements similar to channels of data in the pipeline such as position or texture coordinates. The channel is filled either by underlaying OpenGL at the very beginning of the computation or by another shader if shader aggregation is utilized.

3.1.4.1 Shader Composing

The construction of the Sh and abilities of the C++ language allows a rather straightforward implementation of shader composing [McC04]. Shader composing is basically a attempt for real modularity of the shader thus allowing to create a complex shader based on a functionality of another simpler shader with only a minimal effort and a possibility of applying object oriented programming mechanism.

The basic implementation utilizes a operation overloading to create new set of operators that allows combination of shaders: a **connection operator** denoted as << and a **combination operator** denoted as `&`. The combination operator then allows a process data by both involved shaders sequentially, i.e. the channel generated by the first shader is processed by the second one. The combination operator combine output channels of both involved shaders to a set of channels, i.e. the stream, see below.

In order to improve the performance an optimization is utilized based on dead code removal. Nevertheless, the optimization is not able to distinguish a real computation from a redundant one, e.g. when two shaders are combined and both compute the same value but differ in approach. For such cases an another mechanism has to be employed in order to explicitly denote required path of channel processing, see below.

3.1.4.2 Manipulators and Streams

The possible enhancement of the Sh towards a shader composing with decreased redundancy is a mechanism of manipulators [McC04]. This mechanism allows to operate with individual channels directly by influencing their routing. This improves the overall performance by denoting a dead code and it allows more sophisticated combination of existing shaders.

During the creation phase the mechanism generates a code that provides connectivity between blocks according to a description provided by the application. The description is based on operators mentioned above and a set of manipulators divided into following groups [McC04]:

- **Primitive manipulators** that provide a shader program rather than a stream because they are able to operate with individual elements. It allows keeping and discarding of channels including a possibility to translate elements in the channel via texture mapping, i.e. utilize them as texture coordinates.
- **Fixed-input manipulators** that are able to operate with a fixed number of channels in order to create n copies of channel, pass n channel through that may be utilized during a combination operation, and discard channels.
- **Expandable-input manipulators** that allows manipulating a position of channel thus allowing a redirection of channel to another operation. This may be required when a channel is duplicated and one duplicate has to be utilized as a third input of the block.

Together with manipulator mechanism of streams is utilized. The **stream** is simply an ordered set of channels that uses combining and connecting operator. The combining operator creates a new stream from a two streams and/or channels. The connecting operator connects the stream to a shader.

The shader may be applied to the stream but also it may remain as a shader program with particular inputs connected to channels obtained from the stream. Such inputs are then denoted as parameters. In addition to that the stream mechanism has a capability to convert a parameter back to a an input by a special operator >>. This allows modification of values that were considered to be constants for the whole shader, e.g. k_d coefficient in Phong lighting equation [Wat99].

3.2 General Purposes

No matter how much abstraction is available inside of the code, languages intended for graphical purposes still reflects the GPU structure pretty well. If an application requires general processing on the GPU the solution has to perform the computation in therm of primitive rendering. This is rather annoying because it introduces additional code for bypassing nature of the GPU, which surely complicates the implementation.

On the other hand, there are languages that tries to provide a higher level of abstraction thus hiding the true nature of the GPU and the underlaying API. Also, such languages create a mechanism similar to those of virtual machine where the real implementation is decided at

compilation time according to the nature of requested operation. This simplifies the implementation process. Nevertheless, the performance of such implementation is usually lower then the that of specialized GPU solution due to generality of the language.

3.2.1 Brook for GPU

The Brook [Buc03] is a stream-oriented language intended for stream processing that is developed for specialized high performance stream machines. The pipeline nature of the GPU allows almost direct conversion of the Brook to the GPU with only a few limitations based on features of the GPU thus creating a general tool for implementing general processing algorithms on the GPU.

3.2.1.1 The Language

As it is common among GPU languages, the Brook is based on the C language with only a few enhancements concerning a possibility of streams denoted by `<length>` and kernel parameter specification. **Streams** are essential part of the Brook and represents a homogeneous list of elements of numerical type including arrays and structures. Streams are processed by a kernels.

Kernel is basically a function that process elements of input streams and it computes new values that are stored into an output stream. Besides the input and output streams the kernel is able to access global variables, examine position in the stream, and read other parameters both scalars and arrays that are utilized mostly for gather operation, see below. In common situation, the kernel pushes to the output stream at least as many elements as it is contained inside the input stream. All input elements should have same length for an execution being success.

Code that is inside the kernel function has several limitation that allows its implementation with high throughput or simple parallelization. The fist important issue is a lack of memory management known from the C language including pointers inside the kernel function. Also, library calls as well as non-structural flow controls such as `goto` are not allowed.

Next important feature of the kernel is that for each single execution of the kernel only one value is available from each of input streams. This disables a possibility for the kernel to browse or to look-ahead in the input stream. The execution continues until all data in input streams are processed.

In order to be able to implement various summing operation, the kernel kernel is able to perform a **reduction** operation, i.e. to produce less elements that it is the count of elements in the input kernels. It is possible to reduce either to a single value or to a stream of values where a number of values from input stream reduced to a single element of the output stream is determined from the ratio of input/output stream length.

The Brook language consist of a specialized compiler and a runtime. The runtime provides functionality for executing of kernels, i.e. it distributes data for parallel processing if possible and it is able to generate iteration stream that contains preinitialized sequential values utilized for various gather/scatter operations. It also provides a set high-level stream operators for manipulating stream elements such extracting part of stream or merging streams together.

3.2.1.2 GPU Implementation

As it was already mentioned, thanks to the pipeline of the GPU is it possible to implement the Brook on the GPU thus creating a general tool for general processing on the GPU. Nevertheless,

there are differences in comparison to the Brook specification [Buc03] based on limitations of the GPU.

One of the difference is the maximum length of the stream which is based on maximum size of the texture and numerical accuracy of the particular GPU because the implementation utilizes textures of floating-point elements for indexing a stream. In the case of 1D streams, if such stream would be stored in a form of 1D texture the limitation would be far too restrictive[20].

Therefore, the stream is wrapped over the whole 2D texture utilizing a mapping function that computes texture coordinates according to the index. The index is basically an integer and thus it requires a loss-less conversion when converted to a GPU natural data type such as float, i.e. it requires that all its bits in mantissa are valid. This make length of mantissa[21] crucial for maximum index and thus limiting the stream maximum length.

The GPU implementation supports additional data types common for the GPU, i.e. vectors of length up to four-components are supported. Also, structures are allowed: the implementation utilizes multiple textures, one for each member of particular structure. Data types unsupported by the GPU are not allowed.

The kernel also exhibits few limitations according to the nature of the GPU: it is not possible to perform any operation requiring an inter-kernel synchronization such as writing to a global variable. Actual implementation utilizes the pixel shader for computations and stores a result to a texture by performing a direct rendering to a texture. If such capability is not available[22] the implementation may well utilize various copying operations. The execution is the performed via rendering of a quad over the whole surface of the target texture.

Multiple output streams are supported but they have to be implemented via multiple passes and kernel splitting due to the fact that for a majority of GPUs is able to store only a single four component vector, i.e. they does not support multiple render targets. Despite the fact the dead code optimization for split kernels is utilized the slowdown is almost twice in comparison to single pass approach [Buc04].

Similar to multiple output streams, it is not possible to perform **reduce operation** fully as described in the Brook overview, see previous subsection. Implementation of reduce operation utilizes solution similar to that described in [Krü03] that has a complexity of $O(\log n)$ where n is a ratio of input/output stream sizes, see Subsection 4.4.3.

The performance gained by utilizing the Brook for GPU is a little bit worse than on specialized GPU solution but still better then similar optimized CPU solution[23]. The limitation of performance is caused by multiple passes. Also, the overall performance results may be influenced by limited fillrate and lower texture access bandwidth [Buc04]. Nevertheless, the development time of specialized GPU solution is much longer than that of the Brook and the developer has to deal with the API. For the Brook, the underlaying API is hidden.

20 Currently, the maximum size of a texture for a common GPUs is either 2048×2048 (Ati) or 4096×4096 (nVidia).

21 The maximum length of a number is usually 24 bits according to the floating-point number format s23e8 (nVidia). For some GPU the length is only 17 bits due to lower accuracy of s16e7 format (Ati).

22 This is an issue of the OpenGL prior to OpenGL 2.0 due to lack of render-to-texture mechanism proposed in the specification of `ARB_pixel_buffer_object` [GLE04]. If the mechanism is not available the implementation utilizes a mechanism of pbuffers which is rather slow and bound to a specific operating system platform.

23 For some cases it is possible to gain speedup of up to 7× in comparison to optimized CPU implementation.

3.3 Language Enhancements

3.3.1 Large Code Division

Solutions that utilizes the GPU in order to gain performance in both effect rendering and general processing has to handle several GPU limitations such maximum program length, number of available texture samplers, size of temporary memory, and number of register for vertex shader to pixel shader data transfer.

Even thought some of these limitations may be considered negligible thanks to the improvement of capabilities in newer shader models it can be expected that either some of them remains limited or the complexity of the shaders increases following the improvements. Nevertheless, when the program is overrunning boundaries of limitations the common solution is to use multiple passes, i.e. partition the code to create shorter codes with lower requirement on resources.

Surely, this partitioning can be handled manually but such solution suffers when a boundaries are moved thanks to improvements. Therefore an automatic partition mechanism implemented in a compiler would be handy. A possible solution denoted as recursive denominator split (RDS) is proposed in [Cha02].

The RDS utilizes a directed acyclic graph (DAG) generated during compilation phase. Each node in the DAG represents a hardware-mapped operation, each link then denotes a dependency of various operations. The basic version of the RDS utilizes a merging operation over the DAG together with a heuristic in order to provide a solution that has minimum number of passes. It is assumed that each pass leads to both time and space overhead.

The **merging operation** is handled by a greedy bottom-up algorithm. The algorithm utilizes a construction of **subregion** as a set of all non-marked recursive obtained children of a node *N* including the node *N*. The merging algorithm obtains all direct children of the node *N* to form a list *L*. Form this list all possible subsets *S* of specified length are selected.

For the given subset *S* and the node *N* the algorithm constructs a subregion that is then tested against a possibility of being mapped on a hardware in single pass. If the test is not successful the length of subset *S* is reduced and algorithm continues. If more then one subregion is mappable for given length than a cost-based merge heuristics is utilized.

Also, the algorithm utilizes a heuristics based on special handling of multiple referenced nodes (MR) is utilized, where the MR is a node with multiple parents. The heuristics then decide whether the MR node is recomputed thus saving both space and passes or saved thus saving both computation time and instructions.

In order to partition the DAG properly a partial denominator tree (PDT) is constructed. The PDT is a derivate of DAG and consists only from root, the MR, and partial denominator nodes. A partial denominator node is a node the closest to a particular MR for which is true that all paths from that MR to root goes through such node.

Algorithm that utilizes both merging operation and the PDT is denoted as RDS_h and its flow chart is available in the Figure 3.1. The algorithm divides the DAG according to the PDT thus marking boundaries of node regions. The marking is performed during the merging step in order to denote all nodes that cannot be merged with their parents. The complexity of the algorithm is

$O(n\,g(n))$, where n is number of nodes and $g(n)$ is a cost of checking if set of nodes can be mapped.

```
function RDS_h:
        P' ← root node of the PDT
        SUBDIVIDE(P')

function SUBDIVIDE(T'):
        if subregion(T) is mappable (valid):
                L ← children of T' (order given by postorder traversing of DAG)
                for each K of L:
                        Subdivide(K')
                        if K is MR:
                                use heuristic to decide whether to save or recompute
                apply bottom-up greedy merging to node T
                        mark pixel as processed
```

Figure 3.1: Flow chart of RDS_h algorithm. All variables denoted with apostrophe are performed over the PDT, the rest over the DAG.

The drawback of the RDS_h is the fact that it performs a heuristic based on rather local criteria. It only find one possible combination of regions without any further checking for optimality. In order to bypass that disadvantage the RDS algorithm is introduced.

The RDS performs a limiter search together with cost evaluation to decide which MR node to save and which to recompute. It basically iterates through all MR nodes and for each node M a RDS_h is executed twice. First, for the case the node M is marked, i.e. it has to be saved. Second, for the case the node M in not marked, i.e. it can be recomputed. Afterwards, the algorithm evaluates costs for both version thus deciding if the particular mark of the MR node is fixed or not. The complexity of the RDS algorithm is $O(n^2\,g(n))$.

Both algorithm tries to save number of passes in order to improve the computation time and to utilize GPU computation power. The RDS provides worse performance of $O(n^3)$ in exchange for better partition quality then the RDS_h. Nevertheless, the real performance of the output heavily depends on cost-based heuristics that has to match attributes of particular GPU. For exact measured result of a single experiment, refer to [Cha02].

4 GPU/Hardware Aided Approaches

This chapter contains selected GPU-aided solution of several problem that exhibits a possibility to be implemented on the graphic hardware in order to gain performance. The description is aimed on utilization of the GPU rather than a problem that is solved and thus this chapter should give the reader overview on various approaches for using resources available on the GPU

4.1 Local Illumination

4.1.1 Shading and Materials

Shading and lighting is one the fundamental tasks of the computer graphic that is solved on graphic hardware. Due to the pipeline nature of the graphic hardware, only local illumination is computed, i.e. illumination is solved only for particular point in the space not considering the rest of the scene. This approach has several disadvantages that has to be solved by several emulation techniques. A major disadvantage of the local illumination is a lack of shadow computation whose solutions are introduced in Subsection 4.1.2.

Local illumination based shading is basically computation of a bidirectional reflectance function (BRDF) for given point on a surface in the scene viewer from given viewpoint and lighted by a given light. Currently, there are many of these models that tries to approximate the mechanism [Wat99] but not all of them are simple enough to be implemented with reasonable performance. Surely, it is possible to replace the function by a lookup table but in such case it leads to four-dimensional array that is not available not even in current GPUs as it is mentioned in [Hes99].

Current GPUs allow custom operation per pixel and thus in the case of a simple BRDF the direct implementation is possible. An example of such function that computes shading I_p at given point $\boldsymbol{p}$ for a point light stored in infinity is a Phong model [Pho75] described by the following:

$$I_p = C_p\left[\cos(\alpha)(1-k_d)+k_d\right]+W(\alpha)\left[\cos(\beta)\right]^{\gamma}, \tag{10}$$

where C_p is the reflection coefficient of surface at point $\boldsymbol{p}$ for certain wavelength, k_d is a environmental diffuse reflection coefficient and $W(\alpha)$ is a function of specular reflected light and incident light. Angle α is angle between vector $\boldsymbol{l}$ that points to the light source and normal $\boldsymbol{n}$. Angle β is the vector between vector of reflected light and vector $\boldsymbol{v}$ that points to the viewer. The coefficient γ is the a power of specular reflection.

In order to be able to compute the Equation (10) for each pixel, the GPU has to supply normal for each pixel. A common approach for providing the requested data to the pixel shader is to utilize texture coordinates because they are interpolated over the surface thus providing a estimation of a values for each pixel.

However, the per-component interpolation of normal does not assure that a already normalized vector stays normalized through the interpolation process. If a normal is passed via texture coordinates, it has to be renormalized before use in order to obtain correct results. The **renormalization** is rather an expensive operation because it requires a square root and therefore it may not be available in lower versions of shader. Also, in the newer versions of shader the normalization of vector is time expensive operation thus its faster approximation may be useful.

A possible solution for vector normalization in the pixel shader is to utilize a cube-map texture lookup [Fer03]. With appropriate precomputed textures and linear interpolation sampling it allows to estimate a normalized vector, see Figure 4.1. The approach is compatible even with first versions of pixel shaders.

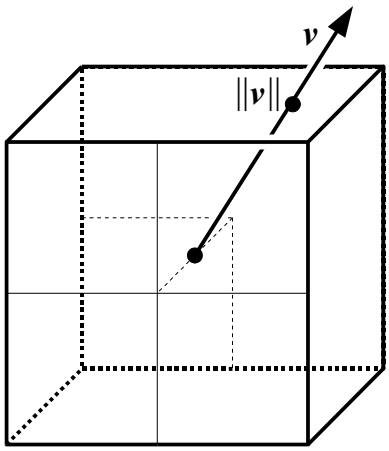


Figure 4.1: Vector normalization via texture lookup.

Another possibility is to constrain the computation. If an orthogonal camera is utilized together with directional lights the renormalization may be omitted completely. In such case, only required dot product may be interpolated over surface instead of vectors. In the [Hei99b] an example of such approach is given for Torrance-Sparrow model in the case of directional light with intensity I_i is described by following:

$$\begin{aligned} f_r &= \frac{F(\cos\theta)\cdot D(\cos\delta)\cdot G(\cos\alpha,\cos\beta')}{\pi\cos\beta}, \\ I_p &= I_i f_r \cos\alpha. \end{aligned} \tag{11}$$

For the Equation (11), consider the ***l*** vector to light, the ***n*** normal of particular surface, the ***v*** vector to viewer, and the ***h*** a halfway vector between ***l*** and ***v***. The function F is a Fresnel term that depends on angle θ that is angle between ***l*** and ***h***. Function D defines a percentage of facets oriented in the direction of ***h*** and depends on angle δ between ***h*** and ***n***. The G is a bivariate Gaussian function that depends on angle α between ***n*** and ***l*** and angle β' between ***n*** and ***v***. The f_r is the BRDF and I_p is intensity at given point on the surface.

Thanks the nature of the Equation (11) components it is possible to implement such lighting model only via texture lookups if the situation meets constrictions mentioned above. The output provides better quality than that of Gouraud and thanks to the missing renormalization the performance is better than that of Phong. Nevertheless, the quality suffers from numerical inaccuracies thus making the result worse than in the case of the Phong.

As it is mentioned in [Hei99b] and [Hei99a] the **anisotropic** shading can be implemented similar to approach mentioned above, i.e. it is possible to enhance the function D to a bivariate function

depending on an angle δ and an angle between facet normal and surface tangent vector. Besides that, it is possible to utilize Banks model that represents an approximation of anisotropic shading. For details on that model, refer to both [Hei99b] and [Hei99a].

Another common operation utilized by many approach is a operation that modifies the rendered surface thus allowing to simulate various small details such as dents or scratches. The modification is only visual, i.e. the geometry and thus silhouette remains the same. One of commonly used techniques that allows to create such effect is a normal mapping.

The **normal mapping** [Fer03, Ern98] is a modification of normal per pixel that is utilized during per-pixel shading according to application supplied texture. The original normal may be only perturbed but in a majority of cases it is replaced completely. This allows to fake the surface in order to hide coarseness of the model, i.e. it allows a model with low polygonal count pretend to be a highly detailed one.

In order to utilize a normal from the normal map during per-pixel computation, the space and other components such as lights has to be transformed to a tangent space. The **tangent space** is defined by an orthogonal basis of original unperturbed normal $\boldsymbol{n}$, tangent vector $\boldsymbol{t}$, binormal vector $\boldsymbol{b}$ for particular triangle. A transformation of point $\boldsymbol{p}$ to a vector $\boldsymbol{v}_{TS}$ in the tangent space with origin in a point $\boldsymbol{p}_s$ can be expressed as matrix-vector multiplication:

$$\begin{aligned} \boldsymbol{v}_{TS} &= \boldsymbol{v}^{\mathrm{T}} \cdot [\boldsymbol{t} \mid \boldsymbol{b} \mid \boldsymbol{n}], \\ \boldsymbol{v} &= \boldsymbol{p} - \boldsymbol{p}_s . \end{aligned} \tag{12}$$

Afterwards, the triangle lies in the of base plane $\boldsymbol{xy}$ and its normal $\boldsymbol{n}_{TS}$ is pointing straight up, i.e. it is basically a vector $(0;0;k), k>0$. This allows a replacement of the normal $\boldsymbol{n}_{TS}$ with a normal loaded from the normal map. For more details, refer to [Ern98, Fer03, Hei99a].

4.1.1.1 Procedural Solid Textures

The drawback of textures as a base of material is a possibility of a pattern to emerge when relatively small texture is mapped to a large surface. Such pattern is rather disturbing because it is a feature that an eye easily catches on. The solution for such surface is to utilize a mechanism common in off-line renderers: a procedural texture.

Procedural texture is a texture defined by a function with surface coordinates as parameters. However, computation of such function for each pixel may introduce visual artifacts due to view dependency of function arguments interpolated over surface. Also, the function may be far too complicated to be computed in a single pass together with other visual effects generated in real-time. Therefore, a mechanism has to be utilized to that allows preprocessing in order to reduce the procedural texture sampling to a texture lookup during object rendering.

A solution proposed in [Car02a] utilizes a known mechanism of mesh atlas and tries to improves the visual quality of the output by employing a mip-mapping and thus reducing aliasing during procedural texture application. Also, it handles artifacts appearing close to a triangle edge that are caused by an application of bilinear texture sampling together with tight packing of triangles in the mesh atlas, i.e. the samples may be disturbed by values from other triangles. The algorithm utilizes following steps that are with exception of the first one performed on the GPU:

1. Unwrap mesh to a 2D texture thus create a mesh atlas, i.e. compute texture coordinates.
2. Store surface coordinates to the texture by rendering unwrapped mesh.
3. Compute surface texture utilizing a precomputed surface coordinates from previous step.
4. Map the texture to real object during real-time rendering similar to regular texture mapping.

The **aliasing issue** is handled by a mip-map mechanism [Wil83] that computes a sample from various sizes of single texture. Sizes are computed by averaging 4×4 texel neighborhood hence triangles have to be reordered to form a 4×4 of neighboring quads, i.e. it utilizes a a modification of uniform grid mesh atlas. The actual position of a single quad withing the quad-tuple is not important because the average operation is commutative.

The construction of such quad-tuples is based recursive application of a algorithm that divides all triangles to two groups. Such algorithm works with a dual representation of a mesh, where a triangle and/or group of triangles is a vertex weighted by size of the triangle and an edge denotes a triangle/triangle group neighborhood relation weighted by a number of edges crossed in the original graph.

The algorithm first utilizes a Metis algorithm [Car02a] of edge collapsing in order to both reduce number of vertices and construct a disjoint partition with balanced sum of graph node weights. After that, the algorithm refines back to original resolution by expansing collapsed vertices and maintaining balance between two partitions. The algorithm is repeated on both created partitions independently. At the end, the algorithm forms group of a up to four triangles that are mapped to a quad in the texture.

Even thought the algorithm utilizes a point-based sampling for each mip-map level and it may lead to oversampling of smaller triangles thus wasting the texture, the overall visual quality is better then for algorithm utilizing weighted mesh atlases. The important advantage of the algorithm is an ability to utilize GPU capabilities for both procedural texture values computation and mip-map generation because newer GPUs as able to generate particular mip-map level on demand.

4.1.1.2 Subsurface Scattering

A subsurface scattering is an effect that allows a lighting to appear in areas completely hidden in the shadow thus resulting in a somewhat blurry surface and shadow, i.e. soft look. For some materials such as skin and marble is this a crucial feature that determines realism of synthetically generated image.

The solution is available in a form of the BRDF generalization into a bidirectional surface scattering distribution function (BSSRDF). The kernel of subsurface scattering is a computation of radiosity b_i in a particular point $\boldsymbol{x_i}$ on a surface S with a knowledge of surface points emittance e_i. If the material properties is defined by the BSSRDF function R_d the radiosity b_i is following:

$$b_i = \int_{x_j \in S} e_j R_d(x_j, x_i) \mathrm{d}\, x_i \,. \tag{13}$$

The computation of the Equation (13) is rather time extensive and thus available only for off-line renderers. Nevertheless, these exist solutions that trades the generality for a performance by restricting features of materials thus allowing almost real-time visualization of such materials.

One of possible solutions for subsurface scattering is described in [Car03] and it is able to compute approximation of diffuse multiple scattering. It utilizes the GPU for computation of a discretization approximation of the Equation (13). The solution utilizes a throughput factor f_{ij} then contains a BSSRDF function R_d and describes a light propagation through the material. The discretization that is the fundamental of the solution is described by following:

$$b_i = \sum_{j=1}^{N} f_{ij} E_j \tag{14}$$

Besides that, the solution utilizes a multidimensional mesh atlas defined in the Subsection 4.1.1.1. The mesh atlas introduces a hierarchy into the mesh and it allows selection of the LOD via the mip-map mechanism. The algorithm requires off-line preprocessing step to generate mesh atlas and during the rendering it requires three passes.

The **preprocessing** step deals with creation of links between patches in order to be able to evaluate the Equation (14). Also, it estimates the throughput factor. First, it builds a mesh atlas and then it utilizes a priority queue to build links between given patch and *l* patches and/or groups of patches with most highest throughput factor by utilizing a mesh atlas hierarchy.

The selection start from the root and it is based on expanding of top of a throughput factor ordered queue. Selection stops when the queue contains *l* links with appropriate throughput factor. Links are then stored to textures, each link containing an uv-coordinate to a mesh atlas texture, throughput factor, and LOD of target patch/patch group.

The **computation** the runs in three steps:

1. Emittance map computation that lights the mesh stored in the mesh atlas texture modulated by the Fresnel term. The output is stored to a texture.
2. Scattered irradiance map generation that utilizes precomputed links and output of previous step. The LOD for a particular link is computed via automatic mip-map generation. As in the previous step, the output is rendered to a texture similar to mesh atlas texture. This is the most time extensive step.
3. Final rendering of object that samples a scattered irradiance map and modulates it by inverse Fresnel term to obtain radiosity of the object outside. The value is further modulated by a common surface lighting forming a color for a given surface pixel.

The **performance** of the solution allows real-time or at least interactive generation of output whose computation time depends on a resolution of the mesh atlas texture and a maximum number of links. The preprocessing step has to be performed off-line because its computation time is out of interactivity boundary. Nevertheless, the preprocessing has to be executed only once because it depends on a shape of the object. Thank to the nature of the last step, the solution may be mixed with other shadowing techniques freely.

4.1.2 Shadows

4.1.2.1 Stenciled Shadows

Shadows are one of the important depth cue known, i.e. they allows estimation of relative objects positions. Shadow may be divided into two categories: soft and hard shadows. While there are many solutions for the hard shadow real-time rendering, the realistic soft-shadow in the real-time is still an area of almost basic research. One of the first solutions for hard shadows able to utilize graphic hardware and therefore able to reach real-time frame rates is a technique based estimation of a shadow volume [Cro77].

Shadow volume is a volume formed by a shadow caster. If a polygon model is rendered it is possible to estimate the shadow volume from the edges of a silhouette visible from the light. Such edge is detected by a dot product of incident faces with a light direction: if signs of dot products are different than the edge is a silhouette edge.

A silhouette edges form a loop of lines that when extruded according to a light direction forms a loop of quads that defines edges of shadow volume. In order to apply a shadow volume to the

scene the algorithm utilizes a **stencil buffer** that allows pixel discarding according to two conditions. Shadows are rendered in following steps:

1. Render scene including depth information in the depth buffer and clear the stencil buffer.
2. Construct a shadow volume mesh for given light and object.
3. Render all back-facing triangle from the shadow volume geometry without modifying the depth buffer. Increment stencil buffer for particular pixel if a depth test fails, i.e. detect pixels that may be inside the shadow volume.
4. Render all front-facing polygon from the shadow volume geometry similar to the previous step. Decrement a stencil buffer if a depth test fails, i.e. omit pixels that are in front of the shadow volume from the viewer viewpoint.
5. Rendering a shadow color quad over the whole scene and modify pixels whose stencil buffer value is larger then zero, i.e. pixels that lie inside the shadow volume.

The algorithm allows real-time hard shadow rendering on the hardware that support stencil buffers. The resulting shadows does suffer from aliasing issues and offers high quality hard shadows along the complete scene. The drawback of the algorithm is its requirement of additional geometry that has to be recomputed whenever a light-object relative position changes. Also, the implementation may suffer from higher requirements on a fill rate and triangle processing performance and the fact that is does not allow reduction of quality for the sake of better performance.

4.1.2.2 Shadow Map

Generation of hard shadows from the geometry described above is a task that is rather computationally extensive because it requires rendering of geometry whose complexity depends on the particular model. Also it has additional requirements on fill-rate capabilities of graphic hardware and requires additional geometry to be computed according to a light-object relative position. The only benefit is that it is able to render precise hard shadows on almost every graphic hardware with stencil buffer support.

Nevertheless, thanks to the programmability the GPU is able to perform various per-pixel operation for shading during which a shadow may be estimated as well according to a shadow algorithm proposed in [Wil78]. The algorithm is rather simple and benefits from a hardware solution of visibility issue: the depth buffer. The hard shadow computation is performed in two steps:

1. Compute a **shadow map**, i.e. depth information for a scene from a viewpoint of the light.
2. Render scene and for each generated point on a surface apply the shadow map by a texture lookup. A point $\boldsymbol{p}$ is transformed to the light space point $\boldsymbol{p}_l(u, v, d)$ and utilized for a shadow map texture lookup in order to obtain a depth of nearest pixel for given position visible from the light. If the depth is higher then $\boldsymbol{p}$ lies in the shadow.

The algorithm is simple and offers a real-time framerates even for complex scene. The drawback of the algorithm is a limited resolution of the shadow map: for an object with high distance from light an aliasing issue occurs. The solution is either to increase the resolution thus moving the aliasing issue further in distance or to utilize hierarchical approach to estimate accurate hard shadow or to limit the light range. Nevertheless, for a majority of real-time application such as games the aliasing issue is ignored for a sake of the performance.

4.1.2.3 Soft Shadows

A hard shadow is a product of lighting by a point light such as bulb or sun. This is usually satisfactory for a majority of scenes but in the case a non-point light such as artificial lighting with a lampshade a soft shadow has to be generated, i.e. a half-shadow has to be added to the edge of hard shadow.

A possible solution for this problem would be to sample the area light with point light and then blend resulting shadows together. This produce reasonable realistic results but is rather computationally extensive because it requires very high number of samples for a single light. Thus, it is not applicable for real-time rendering. Another possibility is to utilize another approach that is able to simulate believable soft shadow.

A solution proposed in [Arv04] emulates a soft shadow by preparing a two lights map for a single shadow caster. The algorithm utilizes basically an enhanced version of shadow mapping with additional geometry and light/shadow maps. In order to provide believable result a half-shadow is divided to a **outer half-shadow** fading from original hard shadow edge to fully lit space and a **inner half-shadow** fading inside the shadow volume.

Each half-shadow part is stored in a separate light map that is computed for each rendered frame by utilizing additional geometry denoted as penumbra quads. During the rendering the algorithm select which light map to utilize according to a **primary shadow map** that is a classical depth information of the scene for the light viewpoint and **secondary shadow map** that is similar to the primary one, but it contains second facing surfaces only.

Penumbra quad is an additional geometry. It is a quad: a wing attached to a copy of every edge on the model and it consist of two triangles. At a setup phase, the every quad has zero size and each vertex contains additional information: a stretching normal estimated as an average of incident faces normals and normals of both faces that shares particular edge that is utilized for silhouette detection. Quads are prepared only once and does not disturb original geometry.

The real-time visualization of soft shadows has basically three streps where the first and the second step has to be performed per single light:

1. Shadow map rendering step during which both primary and secondary shadow map are rendered.
2. Light map rendering step during which both **inner penumbra map** containing inner shadows and **outer penumbra map** containing outer shadow are rendered. This step utilizes penumbra quads and in a vertex shader it computes a distance to which these quads have to be extruded. Only quad attached to silhouette edges are extruded.

   The distance is computed according to the light area and the fact that for far plane of the light source the distance is zero. Besides the distance it also assigns a color coefficient value to vertices according to computed type of the half-shadow, e.g. for outer shadow a quad has to generate values fading from 0.5 (half lit) to 1.0 (fully lit).
3. Scene rendering and soft shadow visualization step that is performed in the pixel shader. Similar to a hard shadow a particular pixel is transformed to a light space and tested against the primary shadow map. If successful the pixel is either fully lit or hit by the outer half-shadow and therefore its color is modulated by the sample from the outer penumbra map.

   The pixel that fails the primary shadow map test is tested against the secondary shadow map. If this test succeeds the pixel is either shadowed or is inside an area of inner half-shadow, i.e.

the inner penumbra map is sample to modulate the color. Otherwise, the pixel is even behind the second layer of light facing surfaces and thus can be considered as shadowed.

Proposed solution is able to deliver realistic soft shadow in real-time for up to two-manifold meshes without high number of passes. Nevertheless, it suffers from several visual artifacts that may disturb the visual quality: aliasing that is caused by a limited resolution of all generated maps and concave objects. The latter one appears when a shadow is casted by an object with deep concave corner thus resulting to a situation that a part of an object is behind a second shadow map. A possible solution may be achieved by dividing the object or increasing the number of shadow maps generated.

4.1.2.4 Subsurface Scattering and Shadow Maps

Shadow mapping is a technique that allows to estimate if a given surface is point in inside the shadow volume or not, i.e. if it should by lighted by the given light or not. Nevertheless, in a reality, the object lighting and shadow does not usually depends only on a particular point and light sources: also other points on the surface contributes to a lighting of the object by a light transfered under the surface, i.e. by subsurface scattering, see Subsection 4.1.1.2.

One of these solutions is proposed in [Dac03]. It is allows computation of materials with high reflectance by utilizing a multiple scattering only because for such materials it is possible to neglect a single scattering and for multiple scattering it is possible to neglect relation of exited and incident light directions.

The solution is theoretically based on a computation of sub-surface scattering that for a given output point $\boldsymbol{x}_{out}$ and set of light incidence points $\boldsymbol{x}_{in}$. The approach is divided into three steps:

1. Computation of incident light intensity $E(\boldsymbol{x}_{in})$ arriving to a surface.
2. Computation of light diffusion $B(\boldsymbol{x}_{out})$ through the material according to a sub-surface reflectance function $R_d(\boldsymbol{x}_{in}, \boldsymbol{x}_{out})$ in a form of integral described by the Equation (15). The equation assumes that there are a direct connection between $\boldsymbol{x}_{in}$ and $\boldsymbol{x}_{out}$ that lies completely inside the objects surface S.

$$B(\boldsymbol{x}_{out}) = \int_S E(\boldsymbol{x}_{in}) R_d(\boldsymbol{x}_{in}, \boldsymbol{x}_{out}) \, \mathrm{d}\boldsymbol{x}_{in} . \tag{15}$$

3. Computation of light radiance from the surface according to diffused light.

The algorithm proposed in the solution is similar to the shadow mapping described above. All computations are performed in the light space in a plane perpendicular to light direction. The algorithm utilizes two steps:

1. Computation of translucency shadow map (TSM) from a viewpoint of a light. For each point $\boldsymbol{x}_{in}$ visible from a light, an arriving light, a surface normal, and a distance from light is computed and stored to a two textures.
2. Apply translucency step that is basically a merge of a second and a third step of the approach described above. First, during the rendering it transforms a given surface point $\boldsymbol{p}$ to light space point $\boldsymbol{p}_l(u, v, d)$ with associated distance from the light source. Such point $\boldsymbol{p}_l$ is then utilized for addressing of shadow map computed in previous step.

Due to performance reasons, the translucency application step is divided into two parts: a local response and global computation. Both these steps evaluates integral in the Equation (15). **Local response** is a basically a contribution of nearest points and thanks to that it is possible to express

it as a weighted sum of 7×7 neighborhood of the point $\boldsymbol{p}_l$. Only texels of specified proximity are taken into account in the neighborhood. If distance of texel at $\boldsymbol{p}_l$ is significantly lower then the distance of $\boldsymbol{p}_l$ then point $\boldsymbol{p}$ is shadowed.

Global response is a contribution of points that are further then points utilized in local response. In this case the integral in the Equation (15) is approximated by a hierarchical filter that performs weighted sums of light contributions in the TSM. For a filter form, refer to Figure 4.2.

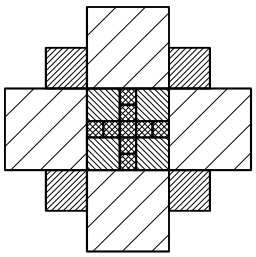


Figure 4.2: Hierarchical TSM filter of 21 samples

The hierarchy of the filter is based on mip-maps generated from the TSM. In order to mask out background pixels from the TSM during the mip-map computation the solution utilizes an alpha channel. After averaging a value of $1/\alpha$ where α is a averaged alpha channel is utilized gain result consisting only from averaged non-background pixels.

Function R_d in the Equation (15) is then approximated by a precomputed texture. However, an original function requires five parameter: a vector $\boldsymbol{r} = \boldsymbol{x}_{in} - \boldsymbol{x}_{out}$ and normal of surface in the point $\boldsymbol{x}_{out}$[24]. Nevertheless, it is possible to reduce the normal to a single coordinate. The second coordinate is then a depth distance between $\boldsymbol{x}_{in}$ and $\boldsymbol{x}_{out}$ and the third has relation to a relative position in the filter[25]. According to the [Dac03] a texture of 64×64×32 is satisfactory.

The solution is able to deliver image with sub-surface scatter at at least interactive rates according to a complexity of the object with a possibility to improve the performance by computing a global response per-vertex utilizing vertex textures. Nevertheless, the solution is not able to take hollows and concavity into account.

4.1.2.5 Shadows of Human Hairs

Human hairs are important feature of human models in both games and virtual reality environments. When applied correctly they increases the visual realism of the image. However, due to their complexity it is not possible to render them in real-time because according to [Kos04] acceptable framerates may not be achieved if more the 10^5 of hair strands are rendered. Also the fact that hairs exhibit anisotropic reflection and self-shadowing makes it nearly impossible to render hairs in per-hair manner as it may be done by off-line renderers.

Therefore, the hairs have to simplified by creating an approximation. In a majority of cases hairs are approximated by a simple solid geometry in order to save computational power but such simplification is not able to deliver realistic images for variety of hair styles. Much better results that exhibit almost all features with aim on shadows of hairs are presented in [Kos04].

24 Normal points toward light and therefore the third component is always positive thus it can be omitted.
25 Second and third coordinate forms together the vector $\boldsymbol{r}$.

The solution proposed in the paper approximates a hairstyles by a configuration of hair wisps. Each hair wisps is geometrically represented as a triangle strip and allows emulation of almost every hairstyle. During the rendering phase, the visual feature of anisotropic rendering is computed in the pixel shader. The solution utilizes normal mapping in order to suggest individual hairs.

Projected shadows as well as self-shadowing is also computed on the fly by utilizing a modified version of **opacity maps**. Depth volume of shadow casting object is evenly divided by *n* parallel planes that are perpendicular to a light direction. To each plane, an alpha-channel image of volume between this and previous plane is rendered. Such plane is denoted as the opacity map. Precomputed opacity maps are then utilized during scene rendering where they act similar to 3D texture, i.e. they define a volume from a viewpoint of the light and values between precomputed planes are estimated by linear interpolation.

The rendering of opacity maps is performed only when a relative position of the shadow caster and the light changes and it is completely done on the GPU. Due to the fact the depth volume of the shadow caster is computed from the bounding box and linear interpolation between opacity maps is utilized, a self-shadowing artifacts may appear. In order to avoid them, it is possible to either increase the density of opacity maps or introduce an offset for opacity map plane position.

The solution allows real-time visualization of hairs even with moving light source exhibiting majority of hair visual features without requirements of additional geometry. The hairstyle approximation allows implementing all possible hairstyles without performance change. The utilized approach for shadowing can be utilized for other situations such a visualization of porous semitransparent material because it is sensitive to light permeability changes in the volume.

4.2 Global Illumination

4.2.1 Ray tracing

Ray tracing is widely utilized global illumination technique in off-line renderers. It allows simple yet powerful control of quality-performance ratio and it is able to visualize many real-world effects such as various atmospheric mirages. All these features are available for off-line rendering due to computational requirements.

Real-time ray tracing is currently available only by utilizing either a massive parallelization or a special hardware. None of both is available for common user due to its cost. However, thanks to the fact that the ray tracing may be performed on a pipeline architecture, a possible solution is to utilize the computation power of the GPU.

For the GPU is programmable, it allows computation of various other values than just pixels. In this case a ray-triangle intersection test may be implemented thus allowing an implementation of probably real-time ray tracer. Nevertheless, such solution would require a GPU→CPU communication that is an usual bottleneck due to its time demands and stalling of the GPU pipeline. A better solution would be to implement a ray tracer completely on the GPU.

4.2.1.1 Ray-Triangle Intersection

One of the possible solutions for the GPU aided ray tracing is a hybrid system described in [Car02b] that utilizes the GPU for ray-triangle intersection. The idea of utilizing the GPU for

such purpose is based on that fact that the intersection is a rather simple operation performed with a large number of elements and thus this makes it ideal for the GPU implementation.

The implementation utilizes the pixel shader for the computation. Rays that has to be intersected are stored in two textures: first for their starting positions and second for their directions. The computation is performed by rendering quads over the area where all vertices of a single quad have triangle data stored in texture coordinates.

The presence of triangle data in each vertex of a quad allows the pixel shader to access same information for each triangle $T(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$ when processing all ray. Each triangle has associated: its positions $\boldsymbol{a}$, $\boldsymbol{b}$, its normal $\boldsymbol{n}$, vectors $\Delta_{ab}=\boldsymbol{a}-\boldsymbol{b}$, Δ_{ac}, Δ_{bc} and color that acts as a triangle ID. In order to save the bandwidth during quad uploading the the GPU an indexed primitive is utilized.

Intersection of a single triangle T with all rays each of origin $\boldsymbol{o}$ and direction $\boldsymbol{d}$ is computed in a single pass according to the Equation (16). If barycentric coordinates u, v, w are non-negative then the ray intersects given triangle. A parameter t denotes a point of intersection for the ray and it is stored in the depth buffer. Thank to capability of the depth buffer test only nearest intersections are kept. ID of such triangle is stored as a color. If possible, the shader may also export barycentric coordinates of the intersection in in order to simplify the CPU processing.

$$\begin{aligned} \Delta_{ao} &= \boldsymbol{o}-\boldsymbol{a}, & t &= -\frac{\boldsymbol{n}^{\mathrm{T}}\cdot\Delta_{ao}}{\boldsymbol{n}^{\mathrm{T}}\cdot\boldsymbol{d}}, \\ \Delta_{bo} &= \boldsymbol{o}-\boldsymbol{b}, & & \\ \alpha &= \Delta_{ao}\times\boldsymbol{d}, & u &= \Delta_{ac}^{\mathrm{T}}\cdot\alpha, \\ \beta &= \Delta_{bo}\times\boldsymbol{d}, & v &= -\Delta_{ab}^{\mathrm{T}}\cdot\alpha, \\ & & w &= \Delta_{bc}^{\mathrm{T}}\cdot\beta. \end{aligned} \tag{16}$$

The GPU aided ray-triangle intersection solution is utilized as a major part of a hybrid ray-tracing system: **The Ray Engine**. The system is a hybrid solution where the GPU performs ray-triangle intersections while the CPU generates rays and triangles for next intersection test. In order to improve performance it utilizes octree that allows to decide whether to utilize the GPU or the CPU for given bucket of rays due to fixed overhead expenses of the GPU.

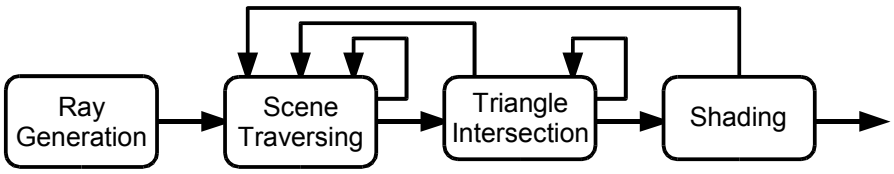
The performance of this hybrid solution is better then the pure CPU solution and it is fully described in [Car02b]. The Ray Engine system is able to benefit from the GPU and the CPU parallelism thus gaining performance improvement. It is also able to optimize size ray textures according to the particular GPU. Nevertheless, the solution is bounded by a fillrate of the GPU and suffers from rather frequent GPU→CPU data transfer.

4.2.1.2 Ray Tracing

A suggestion of possible ray tracer is made in [Pur02]. The solution is able to process scenes constructed from triangles and runs completely on the GPU. The CPU is utilized for control of the computation. Nevertheless, it assumes a specific GPU functionality that may not be available even in the shader model 3.0 but it can be bypassed by a multiple passes.

The solution divides the ray-tracer into four parts and utilizes a stencil buffer for selection which element is processed by which part. Besides that, it utilizes multiple render targets, pixel discarding, and direct stencil write-access in order to perform branching. The last mentioned functionality may be replaced by an additional pass. For a ray-tracer overview, refer to the Figure 4.3a.

a)



b)

```
generate rays:
while any ray is active:
        if ORACLE(rays):
                traverse(rays)
        else:
                intersect(rays)
shade(rays)
```

Figure 4.3: Ray tracer structure (a) and ray tracing algorithm (b).

The implementation utilizes the pixel shader for all operations and the code for the ray tracer parts is available in the Figure 4.3b. The *ORACLE* function decides when to run the intersection. It prevents the intersection to be executed too frequently because execution of that part for every ray when such requirement appears is not effective. Therefore, the intersection is utilized when at least 20 % of all rays are ready for intersection, i.e. they know triangle possibly intersecting them.

The ray tracer parts are:

- Ray generation part that generates rays according to camera settings into a texture.
- Scene traversing part that traverses the scene in order to find triangles that may be intersect by the ray. Based on performance reasons the scene is divided by a regular grid stored as a 3D texture. Each voxel of the grid contains an index to a beginning of indexed triangle list that either are contained inside or intersect the voxel.
- The grid is traversed by the 3D-DDA algorithm and for each pass a single step in the ray traversal is done. When the ray hits a non-empty voxel that contains a link to the triangle list, it is marked so it can be processed in the intersection phase.
- Triangle intersection phase that performs intersection according to marked ray-voxel coupes generated by the previous step[26]. The algorithm utilizes a voxel address together triangle list to address a triangle stored in three texture, a vertex per texture. Each such triangle in the triangle list is tested for an intersection with given ray. Each test takes exactly one pass. If an intersection appears, it is tested against voxel boundary in order to detect intersection outside the voxel and if successful it is passed to the next phase.
- Shading phase process detected intersections similar to a lighting in the hard-wired hardware pipeline. The computation utilizes both output from previous phase and material information associated with triangle to compute the color. If required, shadow rays may be generated. Shadow rays are processed similar to regular eye ray.

Due to the lack of functionality of GPUs at the time the paper was releases the proposed solution was implemented on a software simulator. Various modification of ray tracer were implemented such as a simple shadow caster, Whitted ray tracer, and a path tracer. It is expected that the pure GPU implementation outperforms the CPU version thus allowing a real-time ray tracing on a common personal computer. The implementation also shows that the solution has better performance when flow control is available instead of multiple passes.

26 Marking is denoted by an appropriate value in the stencil buffer.

4.2.1.3 Ray Tracing Enhancements

A pure ray tracer such as Whitted ray tracer [Whi80] provides an images with excellent specular reflection but in the area of diffuse reflection and indirect lighting the approach of ray tracing fails. A possible solution for the latter one is to utilize virtual lights. **Virtual light** is a light derived from a given light source in the scene by shooting random rays from it: whenever the ray touches a surface a virtual light with appropriate intensity is created.

When rendering a virtual light may be threated similar to regular light source. Also, in order to achieve realism, the number of virtual lights has to be unpleasantly high. Based on that, it is sure that the virtual lights mechanism is a feature only for off-line renderers and even for them it may decrease the performance significantly. Therefore, improving the performance of the virtual light evaluation during rendering may have pleasant effect on a total computation time.

A solution described in [Laz04] improves the performance by introducing a heuristic and the GPU utilization for part that has high arithmetic intensity, i.e. the contribution computation. The idea of the GPU use benefits from requirement of virtual light contribution evaluation for every pixel in the image, i.e. a large number of element has to be processed. Also, it benefits from the fact that when sorted by intensity first 50 % of virtual lights are responsible for 90 % of all contribution, i.e. it is possible to omit specific virtual lights first and then continue the lighting computation.

Opposite to the original algorithm, where first the visibility of a light is tested with a contribution estimation computed afterwards, the modified version first evaluates contributions, sorts them, and discards lights of small contribution. Visibility is computed afterwards thus saving the CPU computation time. Contributions are computed in the pixel shader. Contributions of single virtual light source are computed for whole scene in each pass. If possible, multiple virtual lights may be computed in a single pass by packing images to a large render target.

Even though the solution utilizes GPU→CPU data transfer, the performance of the GPU aided solution is magnitude better then those of the CPU because it reduces amount of the light visibility checks. Besides that the solution may perform a partially parallel execution of the CPU and the GPU code.

4.2.2 Radiosity

Another possible global illumination method is a radiosity that is based on a physical model of heat distribution over surface in the scene. It computes a lighting of individual surface elements by estimating a light transfer between two mutually visible elements. In order to fulfill the task a form-factor has to be computed in order to determine influence of individual elements to the rest of elements. The radiosity solution is then a solution equation system defined as follows:

$$b_i = e_i + \rho_i \sum_{j=1}^{N} f_{ij} b_j , \tag{17}$$

where b_i is an element radiosity, e_i is an emittance of the element, ρ_i is a diffuse reflectivity, and f_{ij} is a form-factor, i.e. influence of element j to element i.

A common method for solving of the system depicted in the Equation (17) leads to a rather large equation system. Due to size the equation system it is usually being solved by numerical approaches such as Gauss-Seidel, Jacobi iteration, or others. For matrix solver implementation on the GPU is possible [Krü03], the radiosity may by solved on the GPU as well.

An approach that utilizes Jacobi iteration for radiosity solution is described in [Car03] and allows to compute rather large equation system. The performance of the solution is better the that of the CPU one thanks to parallelism of the GPU: the GPU exhibits almost linear increase of time.

Nevertheless, these exists an iterative approach denoted as **progressive refinement radiosity** that is based on a simulation of light propagation form light sources. The approach requires repetitive computation of numerous elements by applying a rather simple equation and thus it is an ideal candidate for GPU implementation.

A solution that is able to estimate the global illumination of the scene via progressive refinement radiosity is described in [Coo04]. It is able to evaluate radiosity for 10^4 elements in less then second for a scene that consist of planar quads. With the exception of data preparation for both computation and result visualization, the solution run completely on the GPU. An overview of the algorithm is available in the Figure 4.4.

```
set light source quads to particular energy and others to zero
while not converged:
        emitter ← quad with highest residual energy E
        render scene from emitter point of view:
                compute item buffer
                select visible quads according to occlusion query tests
        for every visible quad:
                for every element of such quad:
                        if element visible from the emitter according to item buffer:
                                compute form-factor FF
                                ΔE = ρ · FF · E
                                modify residual by ΔE
                                modify radiosity by ΔE
        set emitter energy E to zero
```

Figure 4.4: Progressive refinement algorithm aided by the GPU.

Each quad requires two sets of radiosity data, namely two textures: the **residual texture** that contains information about an amount of energy that is left to be emitted from the surface and the **radiosity texture**. Due to a limitation of older GPUs, these textures have to be stored either separately thus leading to a multiple passes or packed in a form of 2×16-bits packed to a 32-bits thus requiring an unpack operation before every operation. Nevertheless, the newer GPUs are able to utilize multiple render targets that allows computation of both in a single pass.

A the beginning of algorithm execution depicted in the Figure 4.4 a quad of highest residual power is selected, i.e. quad for which residual energy multiplied by its area is the highest of all quads. Such quads is then considered as an **emitter** that emits its light energy to all elements visible from it. The evaluation of element visibility a fundamental of form form-factor evaluation.

The original CPU-based algorithm utilizes a hemicube erected over the emitter thus requiring computation of five sides of the cube. Such solution leads to multiple passes when copied to the GPU. Therefore the GPU solution utilizes a **stereographic projection** of vertex $\boldsymbol{v}(v_x, v_y, v_z)$ transformed to a emitter view described by following:

$$\begin{aligned} \boldsymbol{v}' &= \boldsymbol{v}/|\boldsymbol{v}|, \\ x &= v_x', \\ y &= v_y', \\ z &= -\frac{2\,v_z + z_{far} + z_{near}}{z_{far} - z_{near}}. \end{aligned} \tag{18}$$

The projection is able to convert all visible space from a particular emitter to a plane and therefore allows visibility estimation. Nevertheless, it also introduces a non-linear deformation to a scene which may cause inaccuracies for larger quads because it transforms only ends of edges that defines straight lines during rendering. In order to approximate the deformation properly, for a nearer and/or larger quads an uniform refinement has to be applied by either a manual refinement at the CPU or N-patches [Vla01].

The **visibility** computation is then similar to a mechanism known from shadow mapping: complete scene from a viewpoint of the emitter is rendered. Each quad is assigned an ID in a form of color that allows identification of pixel affiliation thus forming an **item buffer**. After that a hardware occlusion query is utilized to determine, which quads are visible from the emitter.

For each visible quad a form-factor together with modifications of associated radiosity information is evaluated by rendering to a texture containing new radiosity data. Each element of the quad is transformed to the item buffer according to the Equation (18) and the item buffer lookup is utilized to gather ID of surface that owns elements on that position. If such ID is the ID of currently processed quad, the element is considered visible from the emitter and form-factor is computed.

The computation of form-factor assumes that the value of distance between emitter and particular element is much higher than the size of the element. If the condition is not satisfied a disturbing artifact may appear at adjoining quads such as corners. The possible reduction is based on approximation of element by a disc. This approximation modifies only the form-factor equation described in the paper [Coo04].

After all elements of particular quad were processed, the algorithm has to **select a new emitter** for evaluation. The selection is based on an amount of residual energy and the selected emitter is an emitter with highest amount. The selection is implemented utilizing the depth buffer comparison and rendering to a 1×1 area.

For each quad an average residual power is computed via the lowest level of the mip-map and multiplied by the quad area. Obtained value is utilized as a new depth value of the pixel and the ID of the quad is stored as a color. After all quads were rendered the render target contains the ID of quad with the highest energy that is the new emitter. This step also allows to stop in iterative processing by setting a threshold of highest residual energy.

A solution described in the [Coo04] also introduces a **adaptive** version of the GPU aided radiosity that decreases the aliasing caused by undersampling areas with significant changes in the radiosity. The adaptive solution is based on building of a quad tree over a texture with radiosity data.

After the quad modification is done, the algorithm evaluates a gradient of each radiosity texture element and compares it to a threshold. If the threshold is overlapped the pixel is discarded and occlusion queries are utilized to count the number of such pixels. If the count is high enough a subdivision is performed by creating four child textures of the same size as its parent that each holds an enlarged copy of particular part of its parent.

During the computation each leaf node node of the quad tree is considered as a single quad and it is processed alike. The division maximum depth is limited and thus the child textures may be packed into a single larger textures saving both space and computational power. During the rendering, the geometry has to be refined according to the generated quad tree of particular quad.

The **performance** of the solution is measured and compared to a commercial CPU solution forming of up to 30× faster evaluation of the scene radiosity. Even thought the adaptive version provides a better performance for the CPU-based solution because it avoids oversampling, in the case of the GPU the uniform one is faster. The worse performance of the adaptive version is caused by both frequent context switches such as shader program switching and frequent occlusion queries that stalls the GPU pipeline then in the case of the uniform version.

4.3 Data Visualization

4.3.1 Surface Representation

4.3.1.1 Parallel Occlusion Estimation

Image synthesis is the major purpose of the graphic hardware, which is able to visualize scene in real-time rates. However, for some application such as CAD/CAM the scene may far too complicated beyond the boundaries of real-time visualization due to low filtrate or pipeline bandwidth of the common hardware. In such case a specialized hardware may be the solution. Nevertheless, such specialized is expensive and even such hardware has its limitation. Therefore there are attempts to optimize the scene for visualization purposes at run-time according to a rule that the fastest rendering of a triangle is skipping its rendering.

A possible solution is proposed in [Bax02]. This solution aims primary on a parallel environment and multiple GPUs: it expects two GPUs and one or multiple CPUs utilizing shared memory. The solution makes no assumption about both scene and camera motion thus providing a tool for rendering custom views. The algorithm utilizes following three block of step:

- **Occluder rendering** (OR) block is performed on the first GPU. It computes depth information of the scene according to the visible objects from the previous frame by rendering the geometry without color buffer writes. The gathered information is read back to the CPU accessing memory and passed to the next stage (STC) that is computed completely on the CPU. In order to reduce the waiting time for the STC step, the OR may utilize list of visible objects from last two successive frames.
- **Scene traversal, culling and LOD selection** (STC) block utilizes a hierarchical z-buffer (HZB) [Bax02] that is derived from depth information computed in the OR block. Then, the algorithm traverses the scene tree in postorder manner and performs culling of node bounding box against view frustum and HZB.

  If processed node is visible then a hierarchical level of detail (HLOD) is computed together with pixel error estimation. If the error is acceptable the node is placed into a list of visible node. Otherwise, children of node are processed similar to their parent in order to reduce the error. The visibility list is then passed to RVG for rendering and back to OR for estimation of a next frame depth information.
- **Rendering visible geometry** (RVG) block performs rendering of visible nodes obtained from the STC completely on the second GPU. However, in some cases the GPU may be waiting for the STC. Therefore, the RVG may utilize asynchronous rendering by utilizing possible frame coherency, i.e. starting the rendering of visible geometry from previous frame in advance.

The scene rendered by the proposed solution is stored in a form of node tree containing geometry including precomputed LOD and bounding boxes of node including its children. In a CAD/CAM

models the hierarchy is already present but it follows different goal then an optimal organization for fast visualization.

Therefore, the solution reorganizes the scene by a combined approach of node splitting and merging. At the beginning the algorithm assumes no explicit hierarchy and starts by splitting of larger objects into smaller fragments thus forming new objects. Then, it tries to cluster objects according to their spatial proximity and utilize these clusters as an input for further splitting in order to avoid clusters of uneven shape.

As a final step, the algorithms forms a AABB bounding-box hierarchy based on either center of objects or their group of mass. The tree is utilized for computation of the geometry LOD for leaf nodes and HLOD for inner nodes as combination of its children. The tree is the utilized for both occlusion and LOD estimation.

The solution offers visualization of rather large scene of 10^7 triangle in almost real-time even thought the preprocessing is rather slow. Thank to the nature of the solution it is possible to declare a level of quality in a form of maximum pixel error that influences the performance. For exact results, refer to [Bax02].

The parallel implementation utilizes SGI Onyx with three CPUs and two GPUs. This is rather a special expensive machine but the solution may run on two networked PCs as well. Nevertheless, then next presented solution allows nearly interactive rates on a cluster of common hardware.

4.3.1.2 Distributed Occlusion Estimation

Another possible enhancement of rendering towards a better performance is similar to the previous one, i.e. it tries to improve the performance by culling occluded thus invisible objects. However, in a solution proposed in [Gov03a] the occlusion test utilizes a system with distributed memory connected with Ethernet network.

This solution is based on two fundamentals: frame coherency and occlusion query. Frame coherency assumes that for major means of visualization the view changes rather continuously, i.e. a majority of currently visible object is being visible in next frame too. The occlusion query is then is a hardware feature[27] [Kil04, GLE04] that allows to estimate the number of depth-buffer modifications.

If an application utilizes occlusion query and performs rendering of a bounding-box a two results may appear: either the box is invisible thus everything contained within is occluded or it is visible thus a contained geometry may be visible too. The drawback of the occlusion query is a requirement of GPU→CPU communication that may stall the pipeline. Nevertheless, newer versions of occlusion query may allow accumulation of several queries in order to prevent frequent pipeline stalling.

Rendered scene is a basically a tree of node with a bounding box for each node containing all its children. The solution then utilizes up to three GPU in distributed memory manner. Each GPU has a particular function assigned to:

- **Hardware culling** (HC): Performs occlusion test utilizing the occlusion query and the depth buffer of a current frame. The depth buffer is not modified by the occlusion query test during which the algorithm traverses the scene tree and test bounding boxes of nodes for visibility. If occlusion query fails the node including all its children is considered occluded thus invisible. In the rest of the cases the algorithm considers the node visible.

27 In a case of the OpenGL, the feature is a extension NV_occlusion_query or HP_occlusion_query that allows only detection if depth buffer was modified or not without additional information.

The occlusion query also returns a number of depth buffer modification, i.e. number of visible pixels. This number is then upper bound of all visible pixels from geometry contained within the node and thus it is utilized to compute the LOD of the node. If such number of pixels is higher then defined pixels-of-error value the algorithm process children of the node in similar manner.

All changes in visibility of node is stored to a list that is transmitted over the network to the rest of GPUs in a form of update. This is possible because the algorithm assumes continuous change in the view thus assuming the update list to be much shorter then a complete list.

- **Occlusion representation** (OR): Computes depth information for next frame according to a list of visible objects from a current frame received from the HC GPU. The information is gathered by rendering visible object without color buffer update and it is utilized by the HC GPU in the next frame.
- **Rendering visible geometry** (RVG): Perform rendering of visible geometry for current frame according to a list received from the HC GPU.

In order to limit the communication ration the algorithm switches role of the HC and the OR GPU between successive frame thus allowing the HC to access estimation of scene depth information of current frame computed by the OR without data physical transfer. The important feature of the algorithm is that all three GPUs work in parallel manner with an exception of the first frame where the HC has to wait for the OR to render the depth of the whole. If required, HC and OR may be performed on a single GPU.

The drawback of the algorithm is that it provides the output with latency of at least one frame and it may produces incorrect images due to LOD with an error of up to approximately 20 pixels. Nevertheless, the performance of the algorithm that was estimated by testing it on three models: each with at least 10^7 triangles and at least 10^4 objects, is according to the [Gov03a] better then majority of existing rendering system including Gigawalk [Bax02], see Subsection 4.3.1.1. Also, the solution utilizes only a common hardware rather than special custom systems thus reducing the cost.

4.3.2 Volume Representation

Volumetric data are extension of image data to the space. Through them it is possible to capture a complete sampled image of the scene and similar to image for a plane and thus the volumetric approach suffers from both discretization and low sampling rate due to storage requirements. Nevertheless, opposite to surface representation it allows store a complete information and thus it is ideal for storing structures such as data generated from CT/MRI/3D scanners or it may be utilized as an intermediate result of other representations such as implicit functions or CSG trees.

One of the major problem with volumetric data is its visualization. As opposite to the 2D image, the 3D volume is not suited to be visualized on 2D displaying device and thus a projections is required. There are many methods that allows to project a volume to a plane but only a few of them properly solves occlusion and visibility that are crucial for the visualization.

A possible solution for this task is a conversion of volume data to another representation that may be suited better for the visualization, i.e. an iso-surface extraction. Output of such methods[28] allows utilization of visual quality improving effects with the GPU support but on the other hand they require a long preprocessing step prior to visualization.

28 Marching cubes, marching tetrahedra, and marching triangles.

Another set of method for volume data visualization is based on rendering data directly from the volume. These methods are based on an idea of a ray traversing through the volume and estimation of both density and light collected and/or emitted by the volume. Even though these methods does not produce a surface to which GPU visual effects may be applied, they allow lighting and shadowing thus producing an image of high visual quality.

4.3.2.1 Direct Rendering from Regular Grid Volume

Direct volume rendering allows visualization of volume data without conversion to another intermediate form. Thanks to the fact that they require a rather simple processing of large data set, it is possible benefit from the computation power of the GPU. A description of one such method that allows visualization of data stored in a regular grids is given in [Eng02].

The method is based on blending of a proper selected volume data slices in order to form a volume data projection that may be a projection of required iso-surface as well. The approach described utilizes slices that are perpendicular to the viewer and over these slices it computes a lighting and shadows. On a newer hardware such slices may be computed via 3D textures. Nevertheless, for an older hardware a conversion to set of 2D textures has to be performed.

A simplest approach is to utilize an **axis aligned slice sets**. In order to reduce visual artifacts there has to be three sets of these slices and the renderer has to select a slice set whose axis of alignment is the most perpendicular to the viewer. The drawback is that such selection introduces a popping during set switching.

Nevertheless, it is possible to simulate an effect similar to slicing of 3D texture by creating a set of view aligned slices. Each slice contains a projections of spaces between these slices, i.e. they contains a projection of axis aligned slices. Values from original slices modulated by a value computed from a sample distance to the plane. The creation may be GPU-aided but it requires a multiple pases for a single slice.

Resulting slices are then combined in back-to-front order in a resulting image: an orthogonal projection of a volume. **Perspective** projection of volume data is not trivial but for a mild perspectives an approach similar to the orthogonal projection is possible as well. Nevertheless, for extreme cases, the volume has to be sliced by spherical surfaces that are computationally extensive because they require generation of additional geometry.

The majority of volume data visualization cases displays an **iso-surface** instead of the whole volume. In order to obtain the iso-surface the volume has to be classified by the transfer function. The transfer function is usually an univariate function that can be implemented as a table. The function maps a scalar $s(\boldsymbol{x})$ to a particular color and extinction coefficient according to color density $\tilde{c}(s)$ and extinction density $\tau(s)$.

The classification may be performed either before the slicing denoted as pre-classification or after slicing denoted as post-classification. The first mentioned one does not reproduce high-frequencies on a slice and thus blurs the surface. The latter one reproduces high-frequencies but in order to obtain a visual impression of continuous surface a slice rate has to be increased and thus the performance may suffer.

Nevertheless, it is possible to maintain same number of slices when a **pre-integrated clasification** method is utilized. The method is based on a fact the a the values of the volume are integrated along the path of the ray. The idea is split the integration to a scalar field $s(\boldsymbol{x})$ and the transfer function.

When the segment of the ray is linear between two slices, the transfer function may be reduced to a lookup into a precomputed table. The position in the table is the evaluated from a scalar value of the front slice ray intersection, back slice ray intersection, and length of path segment.

The length may be considered constant and by that the table can be stored as a texture. The assumption of constant length is correct for both orthogonal projection and perspective with spherical slices. For mild perspective, the assumption of constant length provides an acceptable approximation.

During the rendering, values from front and back slice are sampled according to the ray and they are utilized as lookup table coordinates. The pre-integrated classification provides accurate output without increase the slice rate at the same time. The drawback is that the lookup table computation is rather time extensive due to integral evaluation [Eng02].

Besides the proper classification, an important aspect of visual quality of the output is a proper **lighting** and **shading**. In order to perform a shading the approach requires a normal for given volume element computed from its gradients. The computation of the gradient utilizes the first term of Taylor expansion. The gradient I_x for the x-axis is evaluated as follows:

$$I_x(x,y,z)=\frac{s(x+1,\ y,z)-s(x-1,\ y,z)}{2},\quad x,y,z\in\mathbb{N} \tag{19}$$

The lighting computation the normal to estimate the lighting according to equations similar to those in the GPU. In order to save the GPU computation power, the normal may be precomputed and stored in a form of a volume sample. For an older hardware, the lighting may be replaced by lookup into a cube-map with a light information that avoid sampled normal renormalization.

Also, the direct volume rendering allows computation of **shadows** and **self-shadowing**. One of the possible solutions utilizes an approach similar to a shadow map, see Subsection 4.1.2.2. It builds a shadow volume from the viewpoint of the light and utilizes it to modulate values during volume data rendering. The drawback of the method is an appearance of blurry shadows and dark surfaces caused by a coarseness of precomputed shadow volume.

Another approach is based on an accumulation of light that arrives to a particular volume element on each. In ideal case the slices are perpendicular to the light direction but the approach is able to share same the set of slices with the rendering by creating a slices perpendicular to a half-vector between the light and the viewer. If the angle between these two vectors is larger then 90º, the half-vector estimation utilizes a negative view vector instead of the original one.

Before the rendering starts, the algorithm creates a light map initialized to a light intensity. The light map allows creating a visual effects such as a spot light by setting it to an arbitrary image. Then, for each slice it performs two steps:

1. Render slice from the viewpoint modulated by the light map.
2. For each light compute an intensity of light that pass actual slice by rendering a slice from the light viewpoint. During the rendering it utilizes opacity of elements from the current slice to modify the light map, i.e. it accumulates the opacity from the light point of view.

The approach described in the [Eng02] allows visualization of volumetric data in regular grids with interactive and/or real-time frame rates on the GPU. It allows both lighting and shadowing computations in order to improve the visual quality without requirement of any special hardware. Base on a nature of utilized algorithm, it can be expected that the higher version of shader models improves the performance by number of passes reduction.

4.3.2.2 Frequency Domain Rendering from Regular Grid Volume

Frequency volume rendering (FVR) is a method that allows projection of volume data and it is based on a fact that projection of volume in the spatial domain is equal to slicing in a frequency domain. The important feature is the complexity of this process: when the preprocessing phase is omitted the projection has complexity of $O(M^2)$ for $M \times M$ slice.

Such complexity is much better then a complexity of the direct volume rendering for $N \times N \times N$ volume: $O(N^3)$. Also, the approach allows a dynamic control output quality by controlling a slice sampling rather thus allowing to easily implement progressive refinement of the output. Nevertheless, the output of the frequency volume rendering is X-ray-like orthogonal projection of the volume that may not be desirable for viewing purposes.

The algorithm first requires a preprocessing step for transformation to frequency domain by utilizing a Fast Fourier and/or Hartley Transformation with complexity of $O(N^3 \log N)$. Afterwards, transformed data are sliced with a complexity of $O(M^2)$ and transformed back to spatial domain with complexity of $O(M^2 \log M)$ resulting in a X-ray-like image.

The solution described in [Vio04] is a direct implementation of the approach described above on the GPU. The only exception is a **preprocessing** step that has to be performed on the CPU due to the GPU limitations in both 3D texture handling and memory size[29]. During the preprocessing volume is first wrapped around to move the original from a volume corner to the center and then the FFT is applied. Afterwards the result wrapped around again to move the lowest frequencies to the middle of the transformed volume.

During the visualization phase the transformed data are uploaded to the GPU in a form of a 3D texture. **Slicing** is then equal to rendering of a quad perpendicular to the viewer direction centered to the origin of the volume. Proper setting of texture coordinated in corners of the quad then allow data sampling from the volume.

The **sampling** is an important aspect of the method because it greatly influence the visual quality of the result: improper sampling causes ghost-like artifacts to appear in the image. Based on that the solution utilizes a high-order sapling in a form of a filter. The filter is stored as a 1D texture containing weights for four samples and it is applied to each direction separately, i.e. it required three passes. Then, results are summed during the fourth pass thus forming a resulting image.

The last step is an application of an inverse FFT to the slice. The solution utilizes an implementation of GPU-based FFT described in Subsection 4.4.3.3. Before the FFT is applied, data are wrapped around and after the FFT is applied, data are wrapped-around again in order to return the origin to the corner of the image.

The solution provides real-time frame rate during the visualization phase; the preprocessing phase is slower but it has to be performed only once. It has even better performance than the direct volume rendering method and the optimized CPU frequency volume rendering. For the latter one, the reported speed is 17× speed of the CPU version. The drawback of the method is an X-ray-like look of the result. Nevertheless, for larger volumes this method may be the only one capable to provide real-time frame rates on a weaker hardware.

4.3.2.3 Silhouette Extraction from Regular Grid Volume

In a medical area of research but not even in there, a majority of volumetric data are produces by a CT/MRI scans rather than algorithmic approaches based on mathematical description. No

29 For such operation, the video memory have to contain both source and target volume, i.e. $2N^3$ samples.

matter which visualization method is chosen, the resulting image tries to resemble the real-world look. Nevertheless, in some cases the realism is not appropriate because it may hide features that may be crucial also the resulting image is rather complicated even if depth/position clues such as shadows and shading are available. In that case, a non-photorealistic rendering is a solution.

A non-photorealistic rendering allows emphasizing of features or it allows simplification towards better understanding of the image. An important element of the non-photorealistic rendering is a silhouette. It is an important feature of illustrations because it allows the viewer to easily distinguish parts of image from each other.

There are many approaches for silhouette generation from volume data directly. It is also possible to compute the silhouette from intermediate representation such as surface representation but this generally requires greater amount of time then direct rendering. Direct rendering approach then may utilize a gradient but that approach leads to silhouette of various thickness and also it is sensitive to coarseness of the volume data. Another approaches may provide better results but requires significant amount of both space and time.

A solution described in [Nag04] allows extraction of the silhouette of constant thickness and it is able to utilize the GPU for such purposes efficiently thus providing a better performance. The input of the algorithm is a volume sliced according to the viewer direction. Also, the knowledge of which elements are inside or outside the object volume is required in a form of either a already classified volume data or a threshold value for volume data elements.

The algorithm then performs the computation by rendering slices in a front-back order according to a viewer position. This allows extraction of fragments that are hidden if the volume would be rendering and thresholded afterwards. The algorithm utilizes a buffer ***t*** that contains intermediate results. Each pixel of the buffer is labeled and at the begging the label is equal to **empty**, see below.

For each slice, the algorithm sets a label to each pixel to **filled**, if particular voxel is inside the object. Otherwise, it is labeled as **empty**. The classified slice is then merged with intermediate buffer and labels in this buffer are modified to either **empty**, **filled**, **contour** that denotes a possible candidate for silhouette, or **silhouette** according to the scheme in the Figure 4.5.

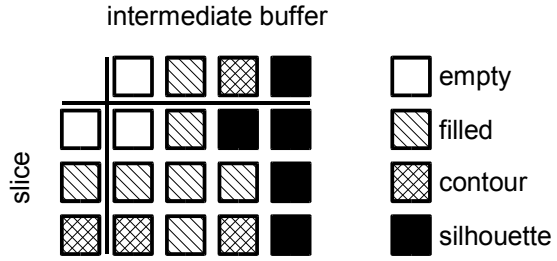


Figure 4.5: Slice × intermediate buffer merging operator.

At the end, the intermediate buffer is thresholded to contain only pixels labeled as **silhouette**. Then it contains results one pixel thick silhouette. Nevertheless, in some application a silhouette of such thickness is not appropriate and thus the silhouette may be broaden by convolution of the intermediate buffer with a Gauss filter:

$$g(x,y)=\frac{1}{2\pi}e^{-\frac{1}{2}(x^2+y^2)}. \tag{20}$$

The resulting image is then thresholded to obtain the silhouette of required thickness. The threshold is determined from the Gauss filter, which is radial symmetric, i.e. it is possible to

substitute $r^2=(x^2+y^2)$. Also, the filter is monotonous in a range $(0;-\infty)$ and thus it is possible to invert the function *g* and thus create a function that provides a conversion from given intensity *h* to distance *r* and apply a threshold:

$$r=g^{-1}(h). \tag{21}$$

The solution allows almost interactive visualization of silhouette even thought the broadening is done on the CPU via Fourier transformation and multiplication of both transformed image and transformed filter. The solution is slower in comparison to gradient based approach but it provides better quality because it is partially insensitive to coarseness of the data.

4.3.2.4 Iso-Surface Computation for Regular Grid Volume

One of the important task for application that utilizes volume data is to be able to segment the data and extract and/or emphasize feature requested by the user, i.e. a proper iso-surface has to be selected defined as a set $S=\{\boldsymbol{x}|\phi(\boldsymbol{x})=0\}$, where function $\phi:\mathbb{R}^3\rightarrow\mathbb{R}$ represents a surface. A simple thresholding of a value may not provide appropriate results due to noise contained within the CT/MRI scans thus requiring manual adjustments.

A possible automatic solution is a solution based on a level-set method that modifies a surface according to a evaluated velocity $\boldsymbol{v}$. For volumetric segmentation the velocity $\boldsymbol{v}$ is replaced according to $\boldsymbol{v}=G(\boldsymbol{x},t)\boldsymbol{n}$, where $\boldsymbol{n}$ is a normal and *G* is a scalar speed of the level set. Hence the modification of the surface can be expressed as a partial differential equation:

$$\frac{\partial\phi}{\partial t}=-|\nabla\phi|G. \tag{22}$$

For purposes of volume data segmentation, the speed term *G* is also influenced by a mean curvature in order to keep surface smooth, i.e. small as possible. The resulting speed term can be expressed as follows:

$$G=-\left[\alpha D(\boldsymbol{x})+(1-\alpha)\nabla\cdot\frac{\nabla\phi}{|\nabla\phi|}\right], \tag{23}$$

where *D* is a function depending of data value at given point $\boldsymbol{x}$ and selected threshold value thus it decides whether the level-set should expand to given element or not. The $\nabla\cdot\nabla\phi/|\nabla\phi|$ is a mean curvature and the α is user selected smoothing term.

The CPU based solution of the level-set is rather slow and it does not provide interactive rates. This is rather unpleasant because the computation requires a lot of proper set up control parameters. Setting of these parameters is rather experimental then a exact estimation and thus an interactivity in ideal situation is appropriate.

A solution that speeds up the process of level-set evaluation is described in [Lef04]. The speedup of the solution is based on utilization of the GPU for the computations. The most interesting part of the solution is a memory management and data sharing system between the GPU and the CPU: the solution utilizes a modification of virtual memory system.

The solution utilizes a **virtual memory system** in order to avoid memory capacity limitations of the GPU and to benefit from a 2D texture as the only output memory element for the GPU, i.e. it is not possible to render to a volume directly. The system is very close to the common virtual memory: memory is divided into 2D pages of 16×16 pixels where a physical page represents a portion of the texture.

Swap-space, mapping tables, and page requests are handled by the CPU. The GPU performs the computation and requests for virtual page at the end of each execution loop. The CPU also handles mapping from the virtual page space to the physical page space be setting of rendered primitive texture coordinates due to the fact that the remapping process would both introduce a dependent texture lookup to the shader code and increase memory requirements for storage of the table.

Based on the fact that the computation follows Equations (22) and (23), the computation kernel requires to access a 3×3×3 neighborhood of each element. If such element is on a boundary of the page, the kernel has to access an adjacent virtual page and it has to store to the result to the same position in different buffer.

Due to lack of scatter operation of the GPU, this has to be handled at the CPU level by rendering a proper primitive: a quad for interior element, a line for boundary element, and a point for corner element. Also, the CPU assures that required adjacent page is uploaded to the GPU before the computation starts.

The computation is performed over **active pages**, i.e. pages that contains a part of a level-set. These pages are processed by the computational kernel and based on performance reason it would be suitable to keep their number low. For purpose the solution tries to maintain constant level-set density by embedding the ϕ function. This is done by introducing a rescaling speed term G_r that ensures gradients further from boundary being zero:

$$G_r = \phi g_\phi - \phi |\nabla \phi| \,, \tag{24}$$

where the g_ϕ is a target gradient magnitude.

During the computation, the solver identifies active pages by inspective their gradient. If a pages contains at least one pixel with non-zero gradient, it is considered as a active page and recomputed. In order to improve the effectiveness of the active page identification process, the data are clamped by a distance transform: outside voxels are black, inside voxels are white.

In order to save the CPU-GPU data transfer, two static pages are created: a page of outside voxels and page of inside voxels. These pages are considered as **inactive pages** and whenever an inactive page is requested as a adjacent page to an active one, a particular static page is utilized instead of uploading a copy.

At a startup time an initial set of active pages is uploaded. The initial set is provided either by the user as an approximation of the level-set or by a threshold. During the computation it is possible to influence the actual level-set in order to gain better results. The level-set evaluation executes sequentially following steps:

1. Computation of 1st and 2nd partial derivate for active pages. Computed values are stored to four temporary buffer to be used during the active page selection.
2. Computation of speed function according to speed terms described at equations (20) and (21).
3. Modification of the level-set according to speed function. For details, refer to [Lef04].
4. Selection of active pages for update according to active page definition, see above. The update utilizes a automatic mip-mapping capabilities of the GPU to obtain an average for gradient components that is thresholded and transformed to a bit-fields denoting pages that are considered active. Bit-wise operations are emulated by additions of $2^k, k \in \mathbb{Z}$ values. This step is executed completely on the GPU.

5. Handling of new active pages and adjacent pages upload that utilizes flag computed in the previous step. The GPU memory then contains current level-set.

In order to allow the user to view the result immediately, the solution utilizes a direct volume rendering approach utilizing a axis aligned slices, see Subsection 4.3.2.1. The evaluated level-set is combined with the original volume on the fly during reconstruction of individual slices and thus it does not utilize a preprocessing step that would involve a reconstruction of the volume. For efficiency reasons, the solution utilizes a gradient precomputed during level-set evaluation for lighting purposes.

The computation performance depends on a number of active voxels. Nevertheless, the solution is able to achieve interactive framerates. When compared to a CPU-based solution, the speedup of the GPU solution is in the order of magnitudes. The solution assumes a sparse level-sets and thus may not be suitable for dense structures. The solution introduces a GPU controlled memory management that saves a computation time even though the creation and processing of the messages requires almost a $\frac{1}{6}$ of total GPU instruction count.

4.3.2.5 Direct Rendering from Tetrahedral Volume

Besides regular grid organization for volume data there are other organization based on unstructured grids. One the of possibilities is a volume composed of tetrahedrons. The visualization of such volume is a task similar to that of structured grid: it is possible to utilize either intermediate form or direct volume rendering. Besides the computation time, the drawback of the intermediate from is a large amount of generated triangles that makes the result inappropriate for interactive visualization.

The direct volume rendering approach utilized for tetrahedral volumes is a modification of an approach possible for structured grid: a splatting of individual volume elements. This approach requires requires a volume elements sorted in correct depth order according to the view direction. Afterwards, individual cells are projected to a screen space thus forming an objects with silhouette of a triangle or a quadrangle.

Each projected cell is then classified according to the Figure 4.6 and mass thickness in the thickest part of the cells is estimated. Afterwards, according to the classification and thickness a geometry consisting of triangles is constructed. Such geometry is then rendered using a regular graphic hardware. If effective, a depth sorting may be performed with plane projections of tetrahedrons instead of with tetrahedron themselves.

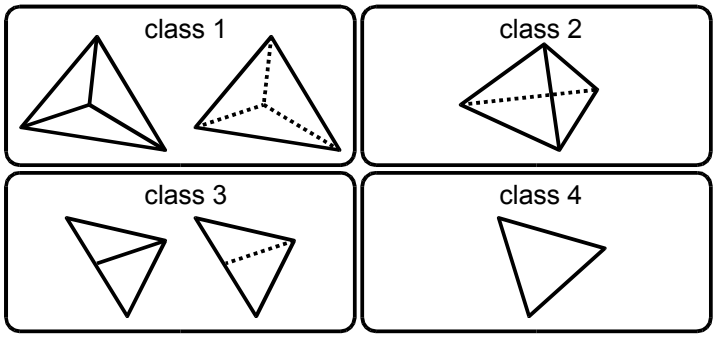


Figure 4.6: Tetrahedral projection classification.

A solution described in [Wyl02] is almost direct implementation of approach mentioned above. With an exception of depth sorting, the solution utilizes the GPU for all steps of the algorithm, in particular the vertex shader that performs the classification, geometry construction, and mass thickness evaluation.

Due to the fact that the vertex shader is not capable to generate additional geometry, the solution has to pass complete required geometry in a form of a four-triangle fan denoted as **basis graph** to the vertex shader and for each passed vertex a complete tetrahedron information has to be added as well. The center vertex of the triangle fan denoted as **phantom vertex** is utilized for emulation of tetrahedron mass thickness.

The algorithm requires a viewer dependent depth sorted list of tetrahedrons. For each tetrahedron a basis graph geometry with complete description of give tetrahedron is passed for rendering. Inside the vertex shader, the geometry is projected to a screen space. Then, tetrahedrons are classified according to the Figure 4.6.

First, it selects a first vertex of tetrahedron, computes vectors to the rest of tetrahedron vertices, and evaluates an intersection point. Then it estimates an order tetrahedron vertices from the viewpoint of selected vertex. The product of the estimation as described in the Figure 4.7 is both identification of particular class and a index to a truth table that provides information required for deformation of the basis graph to tetrahedron projection.

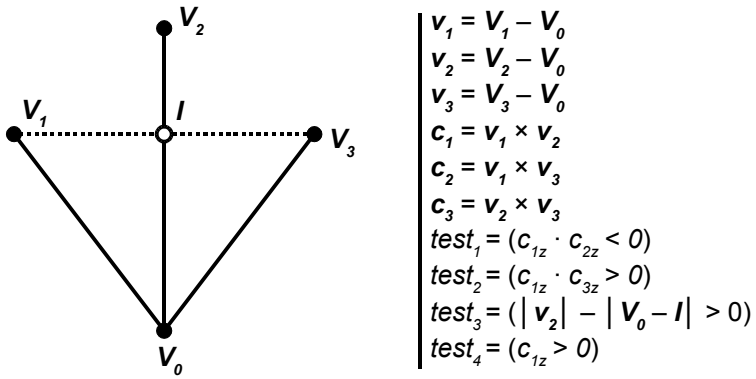


Figure 4.7: Class determination test.

The thickness evaluation depends on particular class. Nevertheless, it is based on thickness at the intersection point ***I*** in the Figure 4.7. The thickness it utilized to modify parameters of phantom vertex such as color and opacity. The rest of vertices has opacity of zero, i.e. they are at the edge of tetrahedron.

The solution provides a possibility to visualize unstructured tetrahedron-based volume data. Even thought the framerates are not interactive, it is expected to be faster then the CPU version for larger data sets. The performance is mostly limited by the CPU→GPU data transfer caused by sorting performed on the CPU. Also, the performance is dragged down by a frequent access to constants, which seems to be even slower then additional shader instructions.

4.3.2.6 Iso-surface Extraction from Tetrahedral Volume

A direct volume rendering approaches allow visualization of the volume with rather high-performance. When using a transfer function they even allow to visualize an estimation of iso-surface utilizing an appropriate transfer function with lighting and shading applied. Also, they allow to compute shadows and thus provide a reasonable visual quality.

Nevertheless, the drawback of these approaches is the fact that they do not reconstruction the iso-surface: they only visualize it. This may be a disadvantage for situation where the transfer function is not changed frequently. Also, these approaches are running in discrete environment and thus may introduce aliasing to the resulting image such as low-resolution shadow map.

A solution described in [Kle04] performs an extraction of the iso-surface from tetrahedral volume and creates a mesh. Both surface extraction and visualization runs completely on the GPU without any GPU→CPU data transfer thus gaining reasonable performance. In order to avoid the data transfer, it utilizes an experimental implementation of extension based on a pixel buffer [GLE04] that makes possible direct rendering to an array of vertices instead the texture.

The feature mentioned above allows the solution to utilize the pixel shader[30] for computation kernel and thus gain performance thanks to its high throughput. Therefore, input data contained in three textures: first two textures are utilized to store the volume and the third one contains a tetrahedron classification. Each corner of the tetrahedron is classified according the value of iso-surface c and its scalar value s thus forming a four-component vector $\boldsymbol{\beta}$, where component β_i is one if c is greater then given s_i and zero otherwise.

The volume is stored in a 2D texture for indices and a 3D texture for vertices, i.e. position and value. This indexed approach allows to reduce the number of floating point values required for a single tetrahedron to 24 in contrary to 80 for non-indexed approach and thus it is possible to upload much larger volume in a limited texture memory. Even thought this approach requires dependent texture lookups it provides much better performance then in a case of CPU→GPU data transfer during the computation.

The output is then a 2D buffer that contains a geometry for intersection of iso-surface and a given tetrahedron. Thanks to the fact that each tetrahedron may produce a quad as the most complicated shape, four pixels stored in a row are utilized. The solution assumes that a quad may degenerate to a triangle without any visual side-effects and quad of zero area is not rendered at all.

This structure allows to store approximately 10^6 quads[31] and thus it is possible to process same amount of tetrahedron in single pass. Also, thanks to this output structure, for given pixel $\boldsymbol{p}(p_x, p_y)$ in the texture, the algorithm can easily determine index of a vertex in the quad as $\upsilon = p_x \bmod 4$. Tetrahedron index to the index texture can be determined as following:

$$\tau = \begin{pmatrix} \lfloor \frac{p_x}{4} \rfloor \\ p_y \end{pmatrix} \tag{25}$$

Computation then utilizes a lookup table in order to obtain which edge determined by two tetrahedron corners $\boldsymbol{V_1}, \boldsymbol{V_2}$ is intersected by the iso-surface for given quad vertex. The table it addressed via the classification vector β[32] and vertex index υ. Resulting vertex is then computed via a linear interpolation $\boldsymbol{V} = t\boldsymbol{V_1} + (1-t)\boldsymbol{V_2}$, where t is evaluated from the scalar value of tetrahedron vertex.

The output are plain quads. Nevertheless, for visualization purposes is it possible to estimate a normal: either per-face from gradient of tetrahedron or per-vertex as interpolation of gradients in tetrahedron vertices. Both of these require additional pass. If multiple render targets are supported, the computation may be performed in a single pass.

Nevertheless, some GPUs may have limitations on number of dependent texture lookups. Thus, it required to reduce the number of these texture lookups. Due to limited texture memory it is not

30 Tetrahedron vs. iso-surface intersection is simple enough and does not exhibit singularities such as those of marching cubes algorithm. This allows to implement the algorithm in a rather length-limited environment of the pixel shader.

31 For a hardware with a capability of handling a render target with maximum dimension of 2048×2048.

32 The classification vector $\boldsymbol{\beta}$ is basically a binary vectors and thus it is possible to handle it as an integer.

possible to remove the indexed nature of the volume. However, it is possible to omit the lookup table utilized for determination of intersection edges by implementing it in the shader.

For **triangle** shape the solution is pretty simple because only a single vertex lies on one side of the intersection and thus vector $\boldsymbol{\beta}$ has only a single component of non-zero value. The second vertex of the edge is determined according to the fact that all intersected edges are growing from the first one. For **quad** shape the situation is more complicated because it leads to a selection of two edge pairs. Nevertheless, even this is possible to express as a set of mathematical operations. For details as well as exact equations, refer to [Kle04].

The **performance** of the solution strongly depends on availability of data in the video memory: if data have to be uploaded during computation, the performance may drop to $1/3$ of original. The visualization of the result depends on performance of actual GPU. In the case of data and/or transfer function change, the solution is able to compute 10^6 tetrahedrons in interactive rates without any sorting or preprocessing done on the CPU.

4.3.2.7 Iso-surface Visualization from Tetrahedral Volume

Another approach for visualization of iso-surface in tetrahedral volume is described in [Pas04]. Similar to the previous approach described in Subsection 4.3.2.6, this one utilizes a conversion to an intermediate form for visualization purposes. However, unlike the previous approach it does store this intermediate form for further visualization: it does both extraction and visualization in a single execution.

As it was mentioned, the output of the algorithm is a triangle mesh and therefore the input has to be a mesh either because the GPU shaders are not capable to generate additional element. In this case, the input is a quad as the most complicated shade that is a result from tetrahedron×iso-surface intersection. The quad is sent in a form of its index in the quad primitive and tetrahedron data such as positions of corners and their scalar values. Index is stored in quad vertex position and associated tetrahedron data are stored in texture coordinates.

The computation kernel resides in the vertex shader because of its capability to both execute longer code and manipulate with individual vertices of render primitive. The computation utilized two lookup tables, both stored in an array of constants. The first table provides an index of four edges that are intersected by the iso-surface. The table is indexed via a classification of the tetrahedron according to an iso-surface value and scalars in tetrahedron corners and these four edges corresponds to four vertices of the quad.

First, the algorithm performs a lookup to the first table in order to obtain four edge. Then, it selects an appropriate edge index according to index of current quad vertex[33]. This edge index is utilized for the second table lookup to provide indices of the tetrahedron corners that defines intersected edge for given quad vertex. Exact position of the quad vertex is then determined via the linear interpolation of obtained vertices. Besides that, the algorithm computes a per-quad normal $\boldsymbol{n}$ from vectors $\boldsymbol{v_i}=\boldsymbol{c_i}-\boldsymbol{c_0}$, where $\boldsymbol{c_i}(x, y, z, s)$ is a corner of the tetrahedron including its scalar value and s_T is a tetrahedron orientation:

$$\boldsymbol{n}=s_T\cdot\left(\det\begin{vmatrix} v_{1y} & v_{1z} & v_{1s} \\ v_{2y} & v_{2z} & v_{2s} \\ v_{3y} & v_{3z} & v_{3s} \end{vmatrix}, \det\begin{vmatrix} v_{1z} & v_{1s} & v_{1x} \\ v_{2z} & v_{2s} & v_{2x} \\ v_{3z} & v_{3s} & v_{3x} \end{vmatrix}, \det\begin{vmatrix} v_{1s} & v_{1x} & v_{1y} \\ v_{2s} & v_{2x} & v_{2y} \\ v_{3s} & v_{3x} & v_{3y} \end{vmatrix}\right) \tag{26}$$

33 This selection is performed as a dot product of edge index vector and quad vertex index bit field, i.e. for third vertex the bit field is a vector (0; 0; 1; 0).

The solution is further improved by utilizing a view dependent division based on division of the longest edge in the tetrahedron. When utilized for regular grid, it produces similar results as octree division and thus allows to control number of generated quad. Nevertheless, this step has to be performed on the CPU.

The **performance** of the solution is hardware dependent. Nevertheless, it is capable to process approximately 10^6 tetrahedrons per second on older hardware. The solution utilize the vertex shader for computation thus making it available with first shader models, i.e. shader model 1.1. However, this also causes lower performance due to the fact that the vertex shader is not build for high throughput.

Further improvement in performance can be achieved by precomputing normals thus saving vertex shader computation time or by utilizing strip of tetrahedrons. The latter one allows to reduce to CPU→GPU data traffic to 40 % or original by sending a modification of one tetrahedron corner instead of all four.

4.3.3 Scattered Data and Point Representations

4.3.3.1 Scattered Data Filtering

Another possible representation of data are scattered data, i.e. point clouds. These data are usually generated by various 3D scanners and coordinate measuring system and utilized as input to various surface reconstruction techniques. Nevertheless, these data usually contains a particular amount of noise that has to be removed prior the reconstruction.

A possible solution for the problems of noise removal is described in [Wan03]. This technique is aimed primary on remove the noise outside the surface assuming that only one object is contained in a set of points. The original solution does not utilize the GPU for the computation but thank to its nature it seems possible to move the solution completely on the GPU.

The idea utilizes for noise removal is based on morphologic operators, see below. In its simplest form, these operators allows removing of independent points or clusters from a binary 2D image. Thus, converting a point cloud to such image would allow removing of these noisy points from the data set.

Morphologic operators are based on operation with sets. If utilized for binary images, input of these operations are sets of points $A=\{(a_{1x}, a_{1y}), ...\}$ and $B=\{(b_{1x}, b_{1y}), ...\}$. If $B_{\boldsymbol{v}}$ denotes a translated set $B_{\boldsymbol{v}}=\{\boldsymbol{x}|\boldsymbol{x}=\boldsymbol{b}+\boldsymbol{v}, \boldsymbol{b}\in B\}$ and $\bar{B}$ denotes a reflection $\bar{B}=\{\boldsymbol{x}|\boldsymbol{x}=-\boldsymbol{b}, \boldsymbol{b}\in B\}$ then Minkowski addition also known as **dilatation** is then defined as $A\oplus B=\{\boldsymbol{x}|(\bar{B})_{\boldsymbol{x}}\cap A\neq\emptyset\}$ and Minkowski subtraction also known as **erosion** is defined as $A\ominus B=\{\boldsymbol{x}|B_{\boldsymbol{x}}\subseteq A\}$.

Based on these operation it is possible to define another two operation: opening and closing. Both are a composition of erosion and dilatation and they differ in their order. The **opening** is defined as $A\circ B=(A\ominus B)\oplus B$ and **closing** as $A\bullet B=(A\oplus B)\ominus B$. The set is usually a rather small set denoted as **structural element**.

In order to obtain such set, the algorithm first projects points to a plane by applying an orthogonal projection in 23 directions: three for major axes and twenty for normals of icosahedron faces. On each of such newly created binary images a morphological operation utilizing a 3×3 morphological element described in the Figure 4.8 is applied to perform operation of opening.

Then, the algorithm identifies a connected set of points and denotes the largest one as the set containing noisy free points because the algorithm assumes that only a single continuous object is available in the input data. Afterwards, the algorithm removes noise points by projecting of original point to all binary images. If an original point is projected to a zero point in a single view, then it is considered a noisy point and removed from the set.

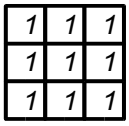


Figure 4.8:Morphological element

The solution is able to remove almost all noise points that lies outside the surface. It can also adjust the surface for removing of fringes possibly caused by the noise. The limitation of the method is the fact that it cannot remove noisy points inside the object volume. The solution utilizes the CPU only. However, thanks to its nature it is possible to move it to the GPU completely including the connected region detection that can be handled by rendering lines in multiple passes, i.e. a modification of coloring algorithm known from computer vision.

4.3.3.2 Point-Based Rendering

Traditional rendering based on displaying of triangles is supported very well in the GPU that are able to render impressive high number of triangles per second. Nevertheless, for large scene, this computation power is not enough and LOD solution is utilized. Still, for distant objects even a coarse triangular mesh is far too detailed and thus a computational power is wasted on unimportant details.

In such case instead of a surface representation a point representation may be utilized without almost any visible artifacts at all. Such hybrid solution is then able to achieve real-time framerates for large walk-through scenes such as urban areas. The only drawback of such solution is the lack of advanced point based representation support in the current GPUs.

The solution described in the [Coc02] benefits from the GPU capability of point sprites and tries to bypass an advanced support required for smooth rendering process such a support of a depth correct order-independent alpha blending. The solution is a hybrid solution for triangle-based models utilizing point representation for lower LODs.

The model is divided by a octree utilizing a **loose LDI tree** (LLDI): a modified version of LDI tree [Cha99] for storing sample points. The LDI tree is an octree with layered depth image (LDI) attached to each node. The LDI is build from a given object by sampling each surface element, i.e. triangle. If two generated points are projected to the same position, the more distant one is stored on a particular position but in the next layer hence the layered depth image.

The LDI tree is the constructed from the triangular model by storing such model into an octree. Then, each leaf contains triangles that are sampled from given viewpoint by orthogonal camera thus forming the LDI. The solution utilizes the GPU rasterizer to sample triangles. Even though it provides results of lower quality then a ray-tracer, it is faster by of a several magnitudes.

Afterwards, the octree is traversed bottom-up computing coarser representation in each inner node according to its children, i.e a lower LOD is constructed. At the end of preprocessing step, three LDI trees exist: each from a view in a direction of an axis. These three LDI tress are merged together to form the LLDI. Besides the position, each point in the LDI contains normal and texture coordinated based on original triangle. Each leaf in the LLDI octree contains also original triangles in order to being able to render higher LODs.

The rendering of the LLDI is the performed according to the Figure 4.9. Triangles are rendered via a common GPU pipeline. Points are rendering utilizing point sprites as a rendering primitives performing a splatting of Gaussian kernel: an alpha is determined from to a Gaussian function and given LOD.

```
for each object in back-to-front manner:
        TraverseOctree(octreeBlock)

function TraverseOctree(octreeBlock):
        if octreeBlock is visible:
                if distance between two adjacent points < threshold d_t:
                        SplatPoints(octreeBlock)
                else:
                        if octreeBlock is leaf:
                                DrawTriangles(octreeBlock)
                        else:
                                for each childOctreeBlock of octreeBlock in back to front order:
                                        Traverse(childOctreeBlock)
```

Figure 4.9: Scene visualization algorithm.

Due to the lack of depth independent alpha blending and the fact that individual splats exceeds area of projected point to fill holes in between, points have to be rendered without a depth test and thus in a correct view-dependent order. This is assured by transversing of given LDI in predefined order. Thanks to a fact that LLDI contains LDI from multiple views, it is possible to bypass visibility overlapping artifacts when an angle between vector to viewer and LDI plane normal is too low, where the **LDI plane** is a plane utilized for projection during LDI setup.

The algorithm utilize the LDI for which the angle is greater than $\pi/4$. Then, an epipolar line is selected. The **epipolar lines** are orthogonal lines in the LDI plane that are both parallel to a particular LDI plane edge. Their intersection is then equal to an orthogonal projection of camera to the LDI plane.

The rendering of given LDI is equal to traversing of the LDI in three nested loops as shown in the Figure 4.10. The outer loop is denoted as *I* traverses the LDI along the second epipolar line in order to avoid visible overlapping artifacts along the first epipolar line where a next iteration of the *I* may disturb previous one close the epipolar line. The first epipolar line divides the LDI and it is possible to traverse each half independently by nester three loops.

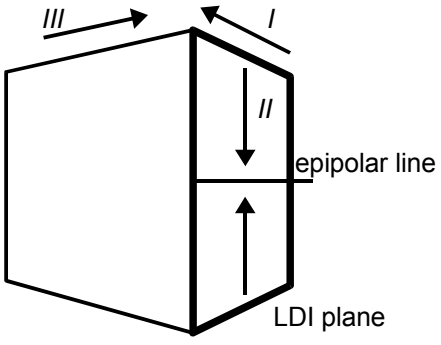


Figure 4.10: The LDI traversing.

The actual LOD is determined as an estimation of maximal distance between two adjacent points thresholded by a threshold d_t. In order to avoid LOD popping when LOD changes from triangular to point-based representation, the algorithm blends both representations. In the case of the LOD change to coarse level between two point-based representations, the algorithm benefits from the fact that the lower LOD contains a subset of higher LOD points. Thus, it is possible to continuously omit points that are not contained in the lower LOD.

The shading is handled by the vertex shader as well as transformation to a position in screen space according to a model-view-projection matrix. Both operations are equal to lighting in the classical hardwired pipeline. Besides that, the vertex shader also computes additional information that is utilized for deforming a Gauss function in the pixel shader according to the view direction.

The Gauss function is stored in a texture utilized from modification of the α-channel of the output and it reflects a given LOD determined by a threshold d_t of a distance between two adjacent points. The threshold then determines the Gauss function for in the distance of $d_t/2$ from the center, the value has to be equal to 0.5. It is implemented via precomputed mip-maps and retrieved alpha is utilized for blending with current background color $\boldsymbol{c}$:

$$\boldsymbol{c} = \alpha \boldsymbol{c}_{splat} + (1 - \alpha)\boldsymbol{c} \tag{27}$$

In order to provide correct image, the kernel has to be rotated according to components n_x, n_y of normal projected to a camera space and $\beta = 1/\cos\theta$, where θ is an angle between view direction and normal. The deformation is the applied by deforming texture coordinate $\boldsymbol{t} = (t_x, t_y)$ by a rotation transformation $\boldsymbol{R}$ based on projected normal components[34]:

$$\boldsymbol{t}' = \beta \boldsymbol{R} \boldsymbol{t}. \tag{28}$$

The solution allows displaying of over 10^5 points at real-time rates for objects of 10^6 triangles. The algorithm is capable to mix both triangular and point representations with smooth LOD changes. This allows effective visualization of large scenes or it allows effective rendering of tiny details on the surface that would require high number of small triangles and thus higher load on the GPU, otherwise.

4.3.4 Other Representations

4.3.4.1 Fast Planar Data Visualization

In a data visualization area there are situations where a huge amount of planar objects is required to be displayed. Example of such area are a geographical application with highly detailed maps, which are far too complicated to be utilized for purposes of high quality real-time visualization even if the solution uses various techniques for visibility culling.

Therefore, there is a requirement for simplification of these planar object to a level depending on particular view, i.e. to remove invisible or nearly-invisible details. This goal can be achieved by removing a vertices thus replacing a polyline segment by a single, i.e. **shortcut**. Nevertheless, the result has to meet several criteria based on geometrical constraints in order to prevent an important features to vanish, e.g. modified polyline should not intersect other polyline if not available in original, polyline should have specified relation to given point even if simplified.

A possible solution for problem mentioned above is proposed in [Mus01]. It allows simplification with preserving of features in real-time with a preprocessing by utilizing a graphic

34 In order simplify and thus speedup the computation it is possible to utilize an approximation described in [Coc02].

hardware. The idea of the algorithm is based on a fact that a shortcut may be considered valid if it is contained completely within a unique neighborhood of a original polyline, i.e. if it does not intersect with neighborhood of different polyline.

The neighborhood of the polyline can be constructed via the Voronoi diagram because areas constructed according to different polyline are unbroken. In order to provide a speedup, it is possible to utilize hardware solution described in Section 4.4.2. Instead of the Voronoi diagram, a **ε-neighborhood** can be used that is defined as a set of points with distance of up to ε from original polyline.

The algorithm then utilizes following steps:

1. Preprocessing step in which the algorithm forms a set of possible shortcuts, i.e. shortcuts that are both of smallest size and of specified maximum distance to original polyline.
2. Construction of the ε-neighborhood to a stencil buffer. For each neighborhood assign a numerical ID.
3. Selection of shortcuts by setting a stencil test and rendering all available shortcuts for given polyline with a unique color ID assigned to each shortcut. Then, read the pixel buffer and look for color pixels, i.e. look for shortcuts that lie outside the neighborhood of given polyline. Such shortcuts may cause disturbance of preserved features and thus they are removed from the list of shortcuts. The rest of the shortcut set is examined again in order to detect shortcut that were overlapped by other shortcuts outside the neighborhood.
4. Construction of shortest path from a set of valid shortcut in order to create a modified polyline. For such purposes a greedy algorithm that selects a shortcut that skips largest number of original polyline vertices and moves to the vertex connected by selected shortcut. The algorithm starts at polyline start and runs until end of polyline is reached.

The proposed solution allows visualization of large planar scenes such as maps with a capability to preserve features such as line-to-line and line-to-point relation. The algorithm is able to deliver the image in real-time with a off-line preprocessing step. The visualization steps utilizes a common GPU and can be improved even further by taking into account of new GPU features such as occlusion query[35] that may greatly speedup the shortcut selecting step of the algorithm.

4.4 Others

4.4.1 Computer Vision and Image Processing

4.4.1.1 Disparity Map Estimation

For a computer vision a depth information from a scene is a crucial because it is required for successful obstacle avoiding. In the case of a real scene, the depth has to be estimated from one or more images. If a pair of images is available, the first step for depth estimation is estimation of a disparity map, i.e. map of pixels with offsets to corresponding pixel in the second map. For such purpose, the images has to compared on a per-pixel basis.

Even though there is the knowledge of camera relative positions that allows to reduce a searching direction and/or radius for each pixel, it still requires a unpleasant high number of independent

35 Occlusion query is available in both OpenGL and Direct3D. For details, refer to [GLE04].

per-pixel operation. That makes the disparity map generation ideal candidate for the GPU implementation because it allows to benefit from parallelism of the GPU.

A solution described in [Zac04] utilizes GPU for all computations and thus it gains an improvement in performance. The solution assumes a stereo pair of images that allows to limit the searching for a scan-line based matching of images. In order to further improve the performance, the solution utilizes a hierarchical approach based on the GPU capability of automatic mip-map generation.

The algorithm starts by comparing a coarser version of images in a rather large range in order to estimate a startup disparity map. Then, it moves to higher level and the comparison step is repeated but in this time, only a limited range based on coarseness difference between levels is utilized, i.e. it refines the output. The disparity map from previous level is utilized for setting a search startup position.

The comparison of images is rather a simple. The algorithm utilize a simple absolute difference of pixel values in order to find a configuration with the lowest sum of absolute difference (SAD) possible. First, it places a window from the second image over the first one and computes an estimation of the SAD by utilizing a bilinear interpolation[36]. Afterwards, the result is blended with actual disparity map by selecting a disparity with the lowest SAD.

At the end, a verification is utilized to discard a possible invalid pixels. For such purpose the algorithm computes a left-to-right disparity map of offsets D_L that is computed by matching a window from left image to the right image and right-to-left map D_R. Then, for each given pixel position $\boldsymbol{p}$ and given threshold ε, the pixel is discarded if it fails following:

$$|D_R(\boldsymbol{p}+D_L(\boldsymbol{p}))+D_L(\boldsymbol{p})|<\varepsilon\,. \tag{29}$$

Thanks to the hierarchical approach, the solution is able to inspect a large range in the left images and it is able to discard invalid matches. It utilizes an absolute difference that does not change the range of value and thus it is possible to utilize the approach for various low-cost mobile devices that are usually capable only of low shader model version. Nevertheless, the solution may fail for images with high frequencies because the average utilized in the hierarchy building process acts as an low-pass filter.

4.4.1.2 Epipolar Geometry Computation

The scene depth estimation utilized by both computer vision or scene reconstruction is based on correspondence of element such as points, lines, or rectangles between two or more images. In a ideal situation it possible to determine corresponding point but in a real situation such point may be only estimated due to noise or lighting conditions. Also, a false correspondence may occur.

Therefore, there has to be an option for manual adjustment of computation towards better accuracy. In order to provide an effective adjusting of the computation, the processing has to be performed at almost interactive rates. However, due to large amount of possibly corresponding points it is not possible to achieve a performance close to a interactivity rate. Also, in order to improve the accuracy usually more then two arbitrary views of known camera parameter are utilized and thus the performance is dragged down even further.

The solution described in the [Rod04] utilized a capability of projective texturing available in common graphic hardware in order to improve the performance. The solution is based on a

36 Bilinear approximation allows to compute an average of four adjacent pixels, i.e. their sum. Thus, the $n\times n$ window may be handled as $½n\times ½n$ windows at the beginning of the sum operation.

correspondence estimation of two points and for such purpose it utilizes an epipolar lines that allow to reduce influence of error caused by noise or slightly incorrect camera settings.

The **epipolar line** is basically a line that is projected to a point $\boldsymbol{c_{ip}}$ in the view I_i while it is projected to a line in the rest of views, i.e. $I_j, i \neq j$. According to that, the epipolar line is a ray projected from a camera position $\boldsymbol{c_i}$ through the point $\boldsymbol{c_{ip}}$. In an ideal case for two views, the epipolar line contains points $\boldsymbol{c_{jp}}$ from the view I_j. In the real situation, almost no point lies directly on the epipolar line in the view I_j and thus an error function has to be applied to estimate which points are probably part of the line.

For multiple views is the situation even more complicated. Candidates points $\boldsymbol{v_{ri}}$ found for each couple r of views $I_i, I_j, i \neq j, i = const.$ may not coincide even thought they denote same position in the original scene. In such case, the correspondence is estimated according to a sum of errors of given candidate $\boldsymbol{v_{ri}}$ on epipolar line for individual views I_j: candidate is projected to view I_j and error is evaluated. Candidates with appropriate low sum of errors are then considered points in the real scene.

The **algorithm** utilized in the solution is a slight modification of epipolar mechanism intended for multiple view. Each ray is rendered to a buffer in a form of a line from utilizing an orthogonal camera perpendicular to such ray. This samples the ray and thus creates candidates that each utilizes an error map texture lookup in order to obtain an error for given view I_j. The error map for given view I_j is basically a distance map where a point value in the map is distance to a nearest extracted point $\boldsymbol{c_{jp}}$. The map may be computed similar to the Voronoi diagram (see Subsection 4.4.2.1).

For a system of two view, the algorithm may stop right now and pixels with error lower than a given threshold are considered as points in the scene. For a multiple view, this projection has to repeated for each of the views $I_j, j \neq i$ utilizing an add blending operation for accumulation of errors. Similar to two view situation, the pixels of low values are then considered as real points in the scene.

However, if a point is occluded from a view I_o then an unpleasant high error value may by generated because such occluded point is probably far too distant from any $\boldsymbol{c_{op}}$ points. Such point may be then considered as a false match even though it is a correct one. In order to bypass it, the algorithm clamps the distance error to a given value.

The solution is able to find scene points for 10^5 epipolar lines with a speed magnitude faster then the CPU thanks to the fact that the approach is similar to regular rendering of primitive. The performance has no linear dependence on number of selected candidates, i.e. samples per ray. Thus, it can provide speed of 70× speed of the CPU-based solution even for older graphic hardware[37].

4.4.2 Computational Geometry

4.4.2.1 Generalized Voronoi Diagram Computation

Voronoi diagram is a structure that provides partitioning of the space and/or plane to cells according to a set of input sites $\{A_1, A_2, \ldots, A_k\}$ and given distance function *dist* that allows to evaluate distance between input site A_i and point $\boldsymbol{p}$; the input site is of arbitrary shape. Voronoi cell $V(A_i)$ for given input site A_i is then defined as follows:

37 It does not use any programmable capabilities. These may be utilized in a point $\boldsymbol{c_{ip}}$ detection.

$$\begin{aligned} dom(A_i, A_j) &= \{\boldsymbol{p} | dist(\boldsymbol{p}, A_i) \leq dist(\boldsymbol{p}, A_j)\}, \\ V(A_i) &= \bigcap_{j \neq i} dom(A_i, A_j). \end{aligned} \tag{30}$$

The Voronoi diagram is utilized for preprocessing of data in order to obtain space partition. Through such partition it is then possible to plan route for robot, evaluate proximity queries, perform location optimization, etc. The major drawback of the diagram is its construction time for generalized versions or input sets that consist of other primitive then points. In such case, edges of cell can be a general high-degree curves.

A possible solution that allows construction of all possible Voronoi diagram generalization without performance drop-down is described in [Hof99]. It utilizes graphic hardware to construct discretization of the Voronoi diagram. Thanks to that it is possible even 3D enhancement of the diagram.

The solution is based on idea of distance function evaluation by rendering of proper geometry. The hardware then handles solution of distance function comparison as described in the Equation (30) by utilizing a depth-buffer test: the color contains ID of given input set that owns particular pixels, i.e. space element. The diagram is then reconstructed by walking along edges in the color image.

In the case of 2D Voronoi diagram, the required geometry is simple and depends in type of input set. Maximum height of such geometry is set to a maximum depth value[38] in order to utilize complete accuracy of the depth-buffer. Base of geometry is resized to possibly influence all pixels in the discretization. Input sets supported by the solution are:

- **Point** for which the geometry has form of a cone. For $M \times M$ discretization of the Voronoi diagram, the base of the cone is a circle with diameter of $\sqrt{2} \cdot M$ with depth set to minimum, an apex of the cone is the position of the point in the plane with depth set to maximum. The density of the mesh depends on selected error level.
- **Line** is replaced by a tent-like geometry in a form of two quads and two point as described above at a line endpoints if a line segment is utilized.
- **Polygon** and **curve** is decomposed to line and points thus utilizing same geometry as described above.

Besides the 2D version, the solution is able to compute Voronoi-based partitioning of 3D space. In such case, the 3D space is discretized by horizontal slices $\alpha : z = z_0$ and each slice is handled independently. As distinct to the 2D version, the geometry[39] is more complicated because it has to consider the distance of given primitive to the slice according to a modified distance function $dist_\alpha(x, y) = dist(A_i, (x, y, z_0))$.

Similar to the planar version, complicated input sets are decomposed, e.g. a triangle is composed of the inside, edges, and points. For these input sets then geometry is following:

- **Inside** of the triangle/polygon influence only an orthogonal projection of the triangle through the space. Thus, it forms triangle or mesh of triangles equal to original polygon with vertices transformed by the distance function $dist_\alpha$. If the polygon intersect the slice it is divided and both parts are handled separately.

38 For depth-test set to greater-or-equal, the value is one. The depth-buffer is initialized to zero.

39 The geometry represents the distance function.

- **Lines** are handled by a geometry of elliptical cone with an apex at intersection point and eccentricity according to the angle between line and the slice. In the case of line segment, the geometry is clipped by two parallel planes each containing respective line segment endpoint that are perpendicular to the slice.
- **Point** distance function forms a hyperboloid.

Besides the 3D enhancement, the solution is also able to compute weighted and farthest-site modifications. **Weighted input set** are handled either by translation along distance axis for additive weights or linear scaling along distance axis for multiplicative weights. In the case of farthest-site modification[40], the solution reverses both depth-buffer initialization and depth-test function.

Due to the discretization, the solution suffers from various errors: **mesh error** caused either by inappropriate approximation of the distance function by the mesh or by inaccuracy of the rasterizer and **combinatorial error** caused by inaccuracy of the depth-buffer. The latter one may cause wrong ID evaluation for given pixel or it may omit details, i.e. small regions. Nevertheless, it is possible to decrease such error by adaptive approach that generates detailed version of area around edges.

The solution allows computation of all Voronoi diagrams without any significant performance penalty caused by complicated distance function or input set. The performance may be improved even further by assuming a maximum distance and thus reducing a number of generated pixels. Nevertheless, the drawback of the solution is possible lower accuracy due to discretization.

4.4.2.2 Collision Detection between Complex Models

Collision detection is an important part of environment and/or physical simulation and in a majority of interactive applications, this is one of the most time consuming part that therefore an appropriate a performance is required. For a simple and/or simplified models this task is easy. For complicated models a hierarchy and heuristics is utilized. Nevertheless, it still requires significant amount of computation that may increases according to complexity of $O(N^2)$[41].

A possible aid is to utilize the GPU and thus convert the solution to its discrete version and solve it in the screen space. These solutions provide better performance then strictly CPU-based one. Nevertheless, they are limited either by type of object because they may require closed object only in order to utilize stencil buffer or by bandwidth of GPU→CPU communication or both. Also, in few cases they are able to compute collision between pairs of objects only.

A solution that tries to avoid limitations mentioned above is described in [Gov03b]. It make not assumption about objects and the scene, i.e. it allows processing of open deformable objects and scene that changes rapidly. It is a hybrid solution that utilizes the GPU for estimation of possible collision and the CPU for computation of exact collision.

The estimation of collision is based on estimation of **potentially colliding set** (PCS) that is basically a set of object who collide or are in a close proximity. The PCS is constructed from a complete set of objects $O_1, \ldots, O_n$ in the scene by pruning of fully visible object from given view. An object O_i is **fully visible** if it is not occluded by both objects $O_1, \ldots, O_{i-1}$ and objects $O_{i+1}, \ldots, O_n$. Fully visible object is also considered as not colliding.

40 Furthest-site Voronoi diagram is a digram of point with negated inequality sign in the Equation (29).
41 This complexity is true for checking each object with each other.

The implementation requires two passes. Before the first pass, it clears the depth buffer, sets orthogonal projection, and sets depth function to less-or-equal. Then, objects $O_1, \ldots, O_n$ are rendered and for each object it detects if it is fully visible. Afterwards, it does the same but with objects in opposite order, i.e. $O_n, \ldots, O_1$. Objects that are fully visible in both pass are excluded from the PCS.

The fully visible test is done via occlusion query[42]: the depth test is disable and depth function is changed to greater-or-equal. Inspected object O_i is rendered with occlusion query to test whether it is fully occluded. If not a single pixel passes the depth test using these negated depth function then no object occludes the object O_i. Afterwards, object is rendered to allow processing of subsequent object.

The approach above is first performed with complete objects to compute first estimation of the PCS. In order the improve the result, the second step test full-visibility on per-object-fragment basis. Each fragment of given object O_i is tested for full-visibility with objects $O_1, \ldots, O_{i-1}, O_{i+1}, \ldots, O_n$. Self-collision is not considered.

In order to further improve the performance, the algorithm first tests axis-aligned bounding boxes (AABB boxes) and then it tests at triangle representation level in each step. This allows to reduce requirements on the pipeline performance and thus allows to obviously non-colliding objects quickly. Also, the AABB orthogonal projection is simple: two quads. This allows to perform the occlusion query together with rendering at the same time.

The **performance** of the solution shows linear dependency on both complexity of objects and its number. It is capable to achieve interactive framerates for with of 10^6 triangles. The major advantage of the algorithm is the fact that it is bounded by a fillrate rather than by a bandwidth whose throughput is not changing as fast as does the fillrate. Also, the solution does not require programmability of the GPU and thus it is possible to utilize it even for older devices.

4.4.3 Mathematics

Numerical mathematics is an important branch of mathematics utilized in the computer science because it allows result estimation in tasks where the analytical solution is either almost impossible or cannot be automated. Numerical approaches usually a rather large number of rather simple operation over individual elements that allows to employ parallelism. Also, the iterative nature of these operation allows to utilize streaming architecture for efficient solving.

According to attributes mentioned above, it can be assumed that for a proper result estimation the running time may be unpleasantly hight and thus either an optimization or a hardware much closer to their attributes is required. A hardware that may serve well for this task is the GPU: in particular the GPU capable of shader model 2.0 or newer.

Such hardware then allows to benefit from hight throughput of the pixel shader with only a few limitations such as lower accuracy that may be either considered negligible or avoided by approaches of numerical mathematics. Also, a communication between the GPU and the CPU in particular GPU→CPU data transfer has to be kept at minimum, i.e. in ideal situation, the method should be implemented completely on the GPU and the CPU should fetch results only.

42 Occlusion query mechanism is described in specifications of extensions `HP_occlusion_query` and `NV_occlusion_query`, see [GLE04].

4.4.3.1 Basic Operators for Linear Algebra

A crucial factor for successful numerical method implementation is an appropriate as well as effective implementation of basic operation and data structures such as matrix and vectors. The proposed solution is described in the [Krü03] and it aims on storing of matrices considering efficient matrix-to-matrix operations.

The basic structure utilized in the numerical mathematics is a **vector**. Even thought this is 1D structure, the actual storage may not utilize 1D texture at all due to its inefficiency and size limitations. Each GPU has its own limitation of maximum texture sizes and thus storing a vector to 2D texture enlarges its maximum size to second power of 1D texture case[43]. Also, GPUs tends to have fast access rate to the 2D texture because it is the most utilize texture form for visualization.

Another important structure is a **matrix**. For a matrix a possible solution is to store it directly to a 2D texture. Nevertheless, this may not be efficient storage for several common matrix operations such as transposing. The more efficient solution is to decompose the $N \times N$ matrix $\boldsymbol{M}$ to N diagonal vectors, where the i-th vector $\boldsymbol{v}^i$ starts at i-th column of the matrix and its elements equal to elements of matrix $\boldsymbol{M} = \{m_{i,j}\}$ according to following:

$$v^i_j = m_{j,(i+j) \bmod N} \tag{31}$$

This structure allows to store large matrices, even larger than the maximum size of the 2D texture. Also, it allows to benefit from performance of the 2D texture accesses and its allows simple implementation of both vector-to-matrix product and matrix transposing. The latter operation requires only reordering of vectors that may be implemented as index shifting directly in the GPU without any additional requirements.

Basic **operations** with these two basic structures such as addition, subtraction, and per-element multiplication are rather simple: a single pass per vector that accesses all elements and computes the result. A slightly complicated is a matrix-to-vector product: $\boldsymbol{b} = \boldsymbol{M} \cdot \boldsymbol{a}$. For a given $N \times N$ matrix this operation requires N passes. Each pass multiplies given diagonal vector element with corresponding element in the vector $\boldsymbol{a}$: for the i-th diagonal vector and its the j-th element, the vector element $a_{(i+j) \bmod N}$ [44] is accessed.

The result in a form of N multiplied diagonal vectors is then summed together to form a single vector $\boldsymbol{b}$. If supported, the summation may be implemented by pixel blending. However, the majority of GPUs is not able to perform the blending on 32-bit floating-point render targets and this multiple passes has to be employed. The performance of the product operation may be further improved by multiplying more than one vector at time up to maximum number of samplers[45].

Another important operation operation with matrices and vectors is a **vector reduction**. This operation allows to sum the vector thus providing a final step for dot product operation[46]. Due to lack of advanced flow control, the implementation of N-component vector reduction requires $\frac{1}{2}\log_2(N) - 1$ passes. In each pass, the 2D texture is divided to four quarters that are merged

43 For current hardware, the maximum size of 2D texture is from 2048^2 up to 4096^2 of elements.

44 The modulo operation may be implemented as proper texture addressing mode, i.e. wrapping of texture coordinates.

45 Exactly, number of samplers minus one: one sampler is utilized for original vector.

46 Dot product first multiplies vectors per-element and then sums the result.

according to a given operation to the left-upper one. For the next pass, the approach utilizes the left-upper quarter instead the whole texture.

For a larger number of numeric tasks such as simulation an input and/or an output form a special modification of a matrix, i.e. the **sparse matrix**. It is sure that these tasks case successfully operate with common dense matrix but utilizing features of sparse matrices usually reduces computation time significantly. Nevertheless, in the case of the GPU, the sparse matrix may not effective in comparison to common dense matrix representation.

In the case of **banded matrix**, the implementation is rather simple: zeroed diagonal vectors are omitted and thus matrix-to-vector product computation is reduced in number of passes. The situation in the case of **random sparse matrix** is little bit complicated. A possible solution is described in the Subsection 4.4.3.2.

A slightly different approach is based on **point sprites**: each point represents a non-zero element in the matrix with color equal the a value. During the product computation, points are transformed in the vertex shader to access proper element in the vector $\boldsymbol{a}$, see above.

Nevertheless, this solution employs the vertex shader that has lower throughput than the pixel shader. Such representation of the matrix is read-only if the GPU is not capable to either utilize vertex textures or proper pixel buffer object, see Subsection 2.2.2.1. Otherwise, the solution has to utilize GPU→CPU data transfer thus degrading the performance.

The **performance** of the GPU solution is much better than the CPU: speed of the GPU matrix-to-vector and vector-to-vector operation is 12—15× speed of the CPU version. The only limitations is the fill-rate and number render target switch operations, i.e. number of passes. Nevertheless, for the real application the speedup is only in the range 2—5 due to additional fixed costs and CPU-GPU communication, especially the GPU→CPU data transfer[47]. Thus, for the performance reasons the GPU computing is not efficient for performing only a single operation followed by an immediate result fetch back to the CPU.

4.4.3.2 Sparse Matrix and Sparse Matrix Solver

One of numerical tasks is a simulation of various processes. In a majority, these simulation leads to partial differential equations (PDE). These may be solved by approaches of numerical mathematics based on iterative process of result refining. According to a previous subsection, it is possible to gain the performance by utilizing the GPU as it is described in [Bol03].

In the case of **unstructured grid** it is possible to utilize the PDE to perform optimization tasks such as mesh smoothing. The input of approach is 2-manifold triangle mesh with n vertices, set of edges $E=\{e_{ij}\}$, where e_{ij} is edge between vertices of indices i, j, and boundary condition $\boldsymbol{f}$. Also, a state of two incident triangles is given in a form of matrix $\boldsymbol{A}$. The goal is to estimate vertex associated degree of freedom $\boldsymbol{x}$ such as position, texture coordinates, scalar value for neighborhood $N(i)=\{j|e_{ij}\in E\}$ of vertex i. The solution is then equal to solution of system $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{f}$:

$$\forall i=1,\ldots,n:\quad a_{ii}x_i+\sum_{j\in N(i)}a_{ij}x_j=f_i\,. \tag{32}$$

If such system is symmetric positive definite then it is possible to utilize a **conjugate gradient** method. This method is an iterative process of finding solution by following of path with th highest gradient in residuum that is initialized from an initial guess of x as $\boldsymbol{r}=\boldsymbol{f}-\boldsymbol{A}\boldsymbol{x}$. For more details, refer to [Bol03, Krü03].

47 For current hardware, the download of 1024×1024 32-bit floating-point RGBA texture costs tens of milliseconds.

Among others, the method utilize a matrix-vector product and vector inner-product. In the case of the latter one, it is possible to utilize approach similar to previous subsection. Also, all vectors are stored in a 2D texture according to same rules as in the previous subsection: vector $\boldsymbol{x}$ is stored as a 2D texture X, vector $\boldsymbol{f}$ is stored as a 2D texture F.

Nevertheless, in the cast of matrix-to-vector product the situation is little bit complicated because the matrix is a **random sparse matrix** and thus utilizing an approach similar to dense matrices may not be efficient enough. Opposite to previous section where sparse matrices were implemented via point render primitive, the approach described in the [Bol03] utilizes a texture-based storage that allows updates of the sparse matrix even on the GPU of shader model 2.0.

The sparse matrix $\boldsymbol{A}$ is decomposed to a texture A containing diagonal a vector of the matrix $\boldsymbol{A}$ and a texture A^* that contains off-diagonal non-zero matrix elements. Elements stored in row order packed tightly, i.e. with no gap. In order to be able to decode the contents of the texture A^* another set of texture is generated: indexing texture R and correspondence texture C.

The texture R contains indexes of row beginnings in the texture A^* and therefore it is close equivalent of vertex neighborhood $N(i)$, see above. The texture C has same layout as texture A^* but instead coefficient it contains addresses of corresponding elements in the texture X. The, computation as described in the Equation (32) can be then expressed as following:

$$\begin{aligned} j &= R[i], \\ Y[i] &= A[i]\cdot X[i] + \sum_{c=0}^{k_i-1} A^*[j+c]\cdot X[C[j+c]]. \end{aligned} \tag{33}$$

If the matrix $\boldsymbol{A}$ depends on values x, the solution allows to update the matrix without downloading result back to the CPU and utilizing approach common for CPU-based solvers, i.e. to compute stiffness matrix. Instead, it performs an update utilizing three additional textures that allow to find vertices vertices for triangles incident on given edge e_{ij} for the off-diagonal elements in the texture A^* and thus it allows to bypass scatter operation required otherwise. Diagonal vector stored in the texture A is handled separately. For more details and exact equations, refer to [Bol03].

The **performance** of the GPU-based approach is better than the CPU-based by coefficient of almost 2× speed of the optimized. The solution is able to perform a single step of the iteration process with real-time framerates. Also, it allows modification of the values in the matrix $\boldsymbol{A}$ without the GPU→CPU data transfer thus enlarging the set of task of possible efficient GPU implementation.

4.4.3.3 Fast Fourier Transformation

Fourier transformation provides a facility to convert from spatial domain to frequency domain. It is based on a fact that it is possible to describe every periodic function as a combination of infinite set of sinus functions. For a non-periodic function, the whole definition domain is considered as a single period. Based on that, the transformation determines correlation of the input function $f(x)$ with set of frequencies in order to form a transformed complex function $F(u)$:

$$\begin{aligned} \mathscr{F}\{f(x)\} &= F(u) = \tfrac{1}{N}\textstyle\sum_{x=0}^{N-1} f(x) W_N^{ux}, \\ \mathscr{F}^{-1}\{F(u)\} &= f(x) = \textstyle\sum_{u=0}^{N-1} F(u) W_N^{-ux}, \\ & W_N = e^{-\frac{1}{N} i 2\pi}. \end{aligned} \tag{34}$$

The drawback of the transformation is its time complexity: for $M \times N$ image it is $O((MN)^2)$. Therefore the most utilized version of Fourier transformation is Fast Fourier Transformation (FFT) that has time complexity of $O(MN(\log N + \log M))$. The algorithm is based on splitting the Equation (34) and input into two halves: half with even indices and half with odd indices. Each half is computed separately thus forming a recursive description:

$$\begin{aligned} F(u) &= \tfrac{1}{N} F'(u), \\ F'(u) &= F'_{even}(u) + W_N^u F'_{odd}(u). \end{aligned} \tag{35}$$

The application of the FFT is in area of loss-compression algorithm and reconstruction of images with en example in the Subsection 4.3.2.2. Also, the FFT may be utilized for improving of filtering performance because a convolution in spatial domain is equal to multiplication in frequency domain. Thus, filtering of $M \times N$ image with $M \times N$ filter in frequency domain is at complexity of $O(MN)$ while the spatial domain requires $O((MN)^2)$.

For GPU-aided solutions, the FFT utilization is problematic because it is performed on the CPU and thus requires a transferring of large data set such as images from the GPU to the CPU. If the FFT would be available on the GPU a possible performance improvement of various methods would be possible by limiting their GPU→CPU communication.

A solution described in [Mor03] is an attempt to implement the FFT on the GPU. It is a almost direct implementation of the FFT as described and for 2D image the forward FFT utilizes two major step: application of the FFT on rows and application of the FFT on columns. The inverse version approach similar to the forward one and at the end the output does not requires additional reordering of the its layout.

In order to convert 2D image to a frequency domain, the algorithm has to perform the FFT described by the Equation (35) on **rows** first. The solution of this steps benefits from the fact that the output of the FFT for input of real numbers are complex numbers that exhibit a certain symmetry $F_{\Re}(u) = F_{\Im}(N-u), \forall u, u \neq 0, u \neq \frac{1}{2}N$. Values of $F(0), F(\frac{1}{2}N)$ does not have a imaginary part.

Based on that symmetry, it is possible to reduce computation time to its half by proper combination of two input rows $r_c(x) = r_1(x) + i\, r_2(x)$. After the transformation, it is possible to decompose the transformed row $R_c(u)$ to individual rows. The symmetry is also utilized to reduce storage requirements by omitting of $\frac{1}{2}N - 1$ complex values thus allowing to store N complex number to N real value slots as described in the Figure 4.11.

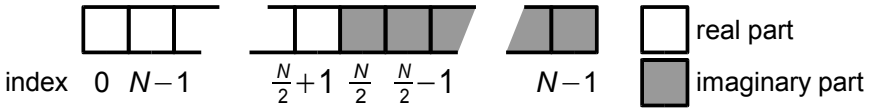


Figure 4.11: The packed complex values generated by the 1D FFT.

The second step in 2D image transformation is to apply the FFT on columns of the the image transformed row-by-row in the previous step. The application of the FFT benefits from a fact that with exception of columns C_0 and $C_{N/2}$ the rest can be merged to a complex numbers. Columns C_0 and $C_{N/2}$ are combined similar to row prior to application of the FFT.

Both described parts of the transformation are implemented in a form of four steps. The inverse FFT utilizes same sequence of step as does to forward one with an exception in step two and four where no scaling is applied. The forward FFT is a sequence of steps:

1. Application of the FFT on rows of the images. Rows are combined as it was described by utilizing upper half of row as real parts and lower half as imaginary parts. Then the FFT is applied in $\log N$ passes according to the Equation (35). Computation of both real and imaginary parts utilizes similar code with difference in indexing.
2. Conversion to packed format described by the Figure 4.11. The conversion perform decomposition of transformed function R_c back to individual transformed rows R_1 and R_2. Then, values are scaled by $\frac{1}{M}$ as described by the Equation (34).
3. Application of the FFT on columns formed by the result from the previous step. Due to the fact that columns C_0 and $C_{N/2}$ contains real parts only they are handled separately similar to rows in first step. The rest of columns contains either real or imaginary parts only and thus they form a complex number when merged.
4. Conversion to packed format similar to the second step. Due to the fact that with only a single exception the composition of two real values to single complex one is not utilized, only scaling is performed. The exception is composition of columns C_0 and $C_{N/2}$ that has to be decomposed similar to the second step.

The **performance** of the solution is under the level of interactivity for image of 1024×1024. It is even slower than optimized CPU-based FFT by a factor of two due to large number of passes and shader code changes. It can be expected that the performance improves when a shader model with dynamic flow control is utilized. Nevertheless, the GPU-based does not require any GPU↔CPU data transfer and thus is may lead to performance improvement for pure GPU solutions.

5 Conclusion

This document provides a view on the GPU and its possible applications. Similar to the graphical hardware previous to the GPU, it outperforms common and comparable CPUs in many tasks. Thank to its capability of being customized by the user supplied code, the area of possible application of the GPU became much larger then it was during previous attempts to utilize graphic hardware. At most, the **general processing** area was affected by the availability of the GPU and the GPU became a tool that allows a performance improvements of several algorithms. This leaded to a explosion of papers among conferences that aimed on the GPU application.

Approaches described in these papers utilized GPU to improve the performance by thinking of it as a second processor available on a machine; some of the even considers the GPU as the dominant processing unit. Nevertheless, both views benefit from the fact that the GPU has high performance, low cost in comparison to dedicated hardware, and it is becoming a standard component in a majority of home computers.

Currently, the GPU utilization for general processing is in the state of an implementation. Based on capabilities and limitations of the GPU it is quite easy to decide whether particular algorithm is suitable for GPU implementation or not and/or to modify particular algorithm to suit the GPU. In a fact GPU application of general processing became a rather a question of conversion between platforms with different limitation and/or runtime requirements than a question of active research aimed on are of possible GPU applications.

When implementing algorithm, for a success it is just required to follow rather simple and intuitive rules that are easily distinguishable from previous GPU applications. Also, it is necessary to keep eye on limitations such as limited flow control capabilities that prevents traversing of sophisticated memory structures and restricted memory access that disallows scatter operations.

Therefore, the GPU is suitable for speedup of brute-force based algorithms that need to process a large number of elements using a simple set of operations. Examples described in this document utilize the GPU for such purposes and their description is aimed on that. Thus, this can be considered as an tutorial on conversion of algorithms to the GPU.

A little bit different area of GPU applications is the area outside the general processing, i.e. area of **graphical applications**. Thanks to the programmability of the GPU it is possible to implement real-time effects that were only available using off-line renderers before. Such effects are then only limited by fillrate, memory latency, and program length issues. In contrast to general processing, this field offers better possibilities for further research but such research is rather restricted because it is tied to a specific hardware configuration, i.e. GPU pipeline.

Besides the GPU application area, it is possible to further investigate the question of the GPU performance improvement based on either its multiplication or multiplication of its parts. The approaches based on **multiplication of GPUs** and/or complete graphic boards requires only a

slight modification of the hardware due to synchronization issues and almost none modification of the GPU itself.

The problems that has to be solved in this area is concerning with a proper data distribution among GPUs over either a bus or a network. The goal is to minimize both redundant work done due to inaccurate distribution and communication required for cooperation of individual GPUs. This also means that the research in this area is highly task dependent in order to improve effectiveness of the solution and thus it rather an impossible task to handle it generally.

A similar situation governs the area of increasing the **number of individual GPU parts**. This is a branch of research that leads to a construction new architectures based on triangle rendering and it is concerned with a solution of work distribution between individual multiplied parts of the GPU. Also, it tries to answer the question of placement of such distribution in the GPU pipeline. This area of research is rather low-level and thus very close to the hardware itself. Theoretical solutions have to be followed by practical experiments with modified hardware in order to estimate the real benefit of particular solution.

The last area of possible research that aims the GPU is an area of **GPU modifications** concerning with introduction of new features rather then multiplying of existing parts. Besides further improvement of the GPU programmability, this field aims on possible modification of the GPU pipeline towards image generation of either better quality, better performance, or both, e.g. input of the pipeline may by modified by support for high-order primitives and effective view-dependent LOD.

Also, the output can be further enhanced by introducing an order-independent transparency support, reduction of anti-aliasing complexity based on view-dependence mechanism, and scatter operation support, i.e. ability to modify target pixel position. Besides that, the inside of the pipeline may be modified as well by introducing a performance improvement such as early visibility estimation in the vertex pipeline or addition of new rendering primitive as it was shown in this document.

In the fact, this is a possible place for real research on the GPU that is not bounded by the solution application and it is not tied to the current architecture as tight as in previous areas. It is even possible to modify the pipeline on a block level and test it on a software emulation in order to measure features such as throughput or fillrate that are easily comparable to possibilities of current and/or future hardware components.

Thanks to that, it is possible to search for improvements without any low-level games with the physical hardware and yet still have a reasonable probability of real implementation in future GPUs. Thus, this is why this area aimed by the future work presented in this state of the art and concept of doctoral thesis.

6 Future Work

Current GPUs are able to synthesize 2D images as a view of 3D scene with very high performance. The reason for generating of 2D images is based on requirements of common display device that is able to display only 2D objects. Currently, thanks to the technology, there is expansion of 3D display devices [Hal97] that either displays a complete volume or tries to reconstruct light waves flowing from the scene and thus recreate almost complete information that can be captured by the human visual system.

Therefore, the question is whether the GPU may be adjusted to fit requirements of 3D display devices. The actual state of the GPU capabilities allows only limited use of the GPU: at majority in similar manner as that of general processing. Such use is not effective because it cannot utilize full power of the GPU that is available in the image generation area. On the other hand, some devices may require a set of images and thus it is possible to utilize the GPU for image generation similar as its developers intended.

The only flaw of this use it that the GPU ignores some information based on the nature of images. That additional information such coherency may then be utilized for improvement of the performance. This is caused by the fact that the GPU does not assume any additional information available even thought such information may reduce the amount of computation significantly such as the case of occlusion tests described in Subsection 4.3.1.

An example that requires series of images is **parallax display device**. This device type is based on discretization of viewing angles, producing a pair of images for each single discretized position. Images are delivered to particular direction by directing of a light thus not requiring head-tracking or similar devices. When user changes a position in front of the device, a different set of images is presented that evokes the feeling of presented scene being actually a solid object, i.e. motion parallax.

Besides that, each eye obtain a slightly different images resulting to a generation of a rather important stereoscopic disparity that evokes feeling of depth. This together with motion parallax and visual realism of individual images, device can convince the viewer that the displayed object is real. The important fact of parallax displays is that there is already a partial technological solution for it that provides full stereoscopic disparity on horizontal motion parallax [Jon95, Yan04] without requirement of user and/or head tracking device.

Based on the nature of this display device type it is possible to utilize the GPU for generation of individual images. The only drawback is the fact, that it requires a set of *2N* images for *N* views at once, e.g. if a framerate of 25 fps[48] is required for 8 view, the graphical device has to generate approximately 400 frames per second. Such high framerate is not achievable by current rendering

48 25 fps is close to a point in framerates where the human visual system can distinguish between set of images and fluent animation.

device if high quality image with proper anti-aliasing is requested because a single GPU solution is bounded by both cost of render target switching and limited fillrate.

A possible and straightforward solution is to multiply the GPU on graphic board level. This solution is possible even with current GPUs and allows to generate images with high performance. The only drawback is its price and requirement of additional hardware device that handles synchronization of output images as well as data distribution between GPUs even thought they do not need to solve the distribution of primitive similar to that of partitioned display [Mit00].

The interesting attribute of the parallax display device is that images required by them are generated from different viewpoints that are located on sphere and looking at a single place. Thus, there is a possibility of coherence in the images. This is basically a fundamental idea of the whole future work that leads to following approach that is close to idea of deferred shading as specified in Subsection 2.2.3.2:

1. Generate all images at once. In ideal situation, only a single rendering pass is required in order to generate both color and depth information.
2. Perform post-processing step with generated image and/or images in order to obtain individual images that can be utilized for the actual output device.

Image generation is the most important part of the whole process. It is based on a fact that when placed in space, individual images from a plane approximation of a sphere or another surface centered at the look-at point, for simplicity let us assume that the surface is spherical. Thus, this approximation of the sphere may be replaced by a real spherical surface that can be unwrapped to a plane without any shearing. Also, there is an inverse transformation back to individual images is possible. This approach is inspired by a sub-step of form-factor computation described in Subsection 4.2.2.

The question that has to be answered is suitability of given surface for these purposes and performing of the mapping. The latter one has to be solved with a respect to a hardware implementation at least from a theoretical point of view because an evaluation of complicated equations as well as operations such division or square root are not acceptable due to both their complexity and performance costs.

Based on knowledge of current GPUs, it can be assumed that a major modification has to done to the rasterizer. It is expected that mapping may be handled by a clever modification of a triangle and/or other primitive through the rasterization thus leaving the rest of the pipeline intact. In the case of the reconstruction, the solution expected a new component at the end of the pipeline: either before data are written to the back buffer or after that. In both cases the solution has to handle situations where from two different view the visibility of two objects changes between each other.

For the output of the solution, there is assumption of a lower quality than the of individual image rendering due to transformation and post-processing step. On the other hand, it is expected that the solution, if possible, improves the performance thank to utilizing of the coherence. Also, if built, the cost of the device should be lower than the cost of brute-force multiplication of individual GPUs.

The solution requires determination of mapping primitive together with solution of linear interpolation on mapped surface as it is common in projection space. There is a possibility to benefit from mathematical evaluation of both mapping transformation and mapping primitive definition that may well help to simplify the interpolation.

The goal of the future work is to modify the GPU, but to left as much as possible unmodified in order to allow compatibility with 2D image generation and integration with other enhancement of classical pipeline as described in Section 2.3. Also, the solution should investigate a possibility of application of this approach to a stereoscopic display device that is able to generate stereoscopic disparity that is a rather important clue when estimating depth and distance. The actual stereoscopic device is available at department of the author.

Also, the solution has to determine a suitable basic primitive that produces lowest amount of inaccuracies when rasterized on non-planar surface, i.e. whether it is still solution to utilize a deformed planar triangle or to switch to spherical one. Besides that, the solution has to determine whether common graphical effects, such as shadows, fog, etc., are still possible utilizing modified GPU or whether there have to be modifications applied. It is expected that the major amount of problems will be experienced when exploring effects that benefit from 2D nature of the output image.

The future work is a part of the 3DTV NoE project that is being solved at the department. This project is aimed on capture, processing, transmission, and display of 3D scenes as well as the 3D space that is the area close to the proposed future work. The major concern of the 3DTV project in an area of displaying is with the holography [Har96] and therefore, the future work will also explore possibility how to incorporate the future work to a process of holography, i.e. basically, how to convert between recorded holographic image and image generated by the solution in order to allow displaying of resulting output utilizing different displays.

Presented future work is a part of research aimed on modification of the GPU. Based on examples described in this work, this is the only place for theoretical research on the GPU. Besides that, it is close to fundamental algorithms of computer graphic because it may happed that a proposed solution based on rendering on non-planar surface may have a slightly different requirements then current rendering to a planar 2D image in order to be effective and thus require either different algorithms or modification of existing algorithms.

References

[Ail03] Aila, T., Miettinen, V., Nordlund, P. (2003). *Delay Streams for Graphics Hardware.* ACM Transactions on Graphics, Vol 22, No 3, pp. 792—800.

[Arv04] Arvo, J., Westerholm, J. (2004). *Hardware Accelerated Soft-Shadows Using Penumbra Quads.* Proc. of WSCG, Pilsen, Czech Republic, pp. 11—17.

[Bax02] Baxter, W. V., Sud, A., Govindaraju, N. K., Manocha, D. (2002). *GigaWalk: Interactive Walkthrough of Complex Environments.* Proc. of the Eurographics Workshop on Rendering, Pisa, Italy, pp. 203—214.

[Bol03] Bolz, J., Farmer, I., Grinspun, E., Schrooder, P. (2003). *Sparse matrix solvers on the GPU: conjugate gradients and multigrid.* ACM Transactions on Graphics, Vol 22, No 3, pp. 917—924.

[Buc03] Buck, I. (2003). *Brook Spec v0.2.* [WWW] http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf (Frebruary 5 2005).

[Buc04] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalain, K., Houston, M., Hanrahan, P. (2004). *Brook for GPUs: Stream Computing on Graphic Hardware.* ACM Transactions on Graphics, Vol 23, No 3, pp. 777—786.

[Car02a] Carr, N. A., Hart, J. C. (2002). *Meshed atlases for real-time procedural solid texturing.* ACM Transactions on Graphics, Vol 21, No 2, pp. 106—131.

[Car02b] Carr, N. A., Hall, J. D., Hart, J. C. (2002). *The ray engine.* Proc. of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Saarbruecken, Germany,pp. 37—46.

[Car03] Carr, N. A., Hall, J. D., Hart, J. C. (2003). *GPU algorithms for radiosity and subsurface scattering.* Proc. of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, San Diego, USA, pp. 51—59.

[Cat80] Catmull, Ed, Smith, E. R. (1980). *3D-Transformations of Images in Scanline Order.* Computer Graphics, Vol 14, No 3, pp. 279—285.

[Cha99] Chang, C., Bishop, G., Lastra, A. *LDI tree: a hierarchical representation for image-based rendering.* Proc. of the 26th conf. on Computer graphics and interactive techniques, Los Angeles, USA, pp. 291—298.

[Cha02] Chan, E., Ng, R., Sen, P., Proudfoot, K., Hanrahan, P. (2002). *Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware.* Proc. of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Saarbrucken, Germany, pp. 69—78.

[Coc02] Coconu, L., Hege, H. (2002). *Hardware-accelerated point-based rendering of complex scenes.* Proc. of the Eurographics Workshop on Rendering, Pisa, Italy, pp. 43—52.

[Coo84] Cook, R., L. (1984). *Shade Trees*. Computer Graphics, Vol 18, No 3, pp. 223—231.

[Coo04] Coombe, G., Harris, M. J., Lastra, A. (2004). *Radiosity on graphics hardware.* Proc. of the 2004 conf. on Graphics interface, London, Canada, pp. 161—168.

[Cro77] Crow, F. C. (1997). *Shadow algorithms for computer graphics.* Proc. of the 4th conf. on Computer graphics and interactive techniques, San Jose, USA, pp. 242—248.

[Dac03] Dachsbacher, C., Stamminger, M. (2003). Translucent shadow maps", Proc. of the Eurographics Workshop on Rendering, Leuven, Belgium, pp. 197—201.

[Die99] Dietrich, S. (1999). *Guard Band Clipping.* Game Developer Conference.

[Die04] Diepstraten, J., Weiskopf, D., Kraus, M., Ertl, T. (2004). *Vragments and Raxels – Relocatability as an Extension to Programmable Rasterization Hardware.* Proc. of WSCG, Pilsen, Czech Republic, pp. 181—187.

[Dog00] Doggett, M., Hirche, J. (2004). *Adaptive View Dependent Tessellation of Displacement Maps.* Proc. of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Interlaken, Switzerland, pp. 59—66.

[DX05] *Microsoft DirectX 9.0 SDK (February 2005) Documentation.* (2005). [WWW] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/directx9cpp.asp (March 17 2005).

[Eng02] Engel, K., Ertl, T. (2002). *Interactive High-Quality Volume Rendering with Flexible Consumer Graphics Hardware.* Proc. of Eurographics, Saarbrucken, Germany, State of the Art Report.

[Ern98] Ernst, I., Russeler, H., Schulz, H., Wittig, O. (1998). *Gouraud bump mapping.* Proc. of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Lisbon, Portugal, pp. 47—ff.

[Fer03] Fernando, R., Kilgard, M. J. (2003). *The Cg Tutorial.* Addison-Wesley.

[GLE04] *OpenGL Extension Registry.* (2004). [WWW] http://oss.sgi.com/projects/ogl-sample/registry/ (March 17 2005).

[Gov03a] Govindaraju, N. K., Sud, A., Yoon, S., Manocha, D. (2003). *Interactive visibility culling in complex environments using occlusion-switches.* Proc. of the 2003 symposium on Interactive 3D graphics, Monterey, USA, pp. 103—112.

[Gov03b] Govindaraju, N. K., Redon, S., Lin, M. C., Manocha, D. (2003). *CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware.* Proc. of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, San Diego, USA, pp. 25—32.

[Hak97] Hakura, Z. S., Gupta, A. (1997). *The design and analysis of a cache architecture for texture mapping.* Proc. of the 24th symposium on Computer architecture, Denver, USA, pp. 108—120.

[Hal97] Halle, M. (1997). Autostereoscopic displays and computer graphics. ACM SIGGRAPH Computer Graphics, Vol 31, No 2, pp. 58—62.

[Han90] Hanrahan, P., Lawson, J. *A language for shading and lighting calculations.* Proc. of the 17th conf. on Computer graphics and interactive techniques, Dallas, USA, pp. 289—298.

[Har96] Hariharan, P. (1996). *Optical Holography*. Cambridge University Press.

[Hei99a] Heidrich, W. (1999). *High-quality Shading and Lighting for Hardware-accelerated Rendering.* Ph.D. thesis, University of Erlangen, Erlangen, Germany.

[Hei99b] Heidrich, W., Seidel, H. (1999). *Realistic, hardware-accelerated shading and lighting.* Proc. of the 26th conf. on Computer graphics and interactive techniques, Los Angeles, USA, pp. 171—178.

[Hof99] Hoff, K. E., Keyser, J., Lin, M., Manocha, Culver, T. (1999). *Fast computation of generalized Voronoi diagrams using graphics hardware.* Proc. of the 26th conf. on Computer graphics and interactive techniques, Los Angeles, USA, pp. 277—286.

[Hor04] Hormann, K., Tarini, M. (2004). *A Quadrilateral Rendering Primitive.* Proc. of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Grenoble, France, pp. 7—14.

[IEE85] 754:1985. *IEEE Standard for Binary Floating-Point Arithmetic*.

[Jon95] Jones, M. W., Nordin, J. H., Kulick, J. H., Lindquist, R. G., Kowel, S. T. (1995). Real-time three-dimensional display based on the partial pixel architecture. Optical Letters, Vol 20, No 12, pp. 1418—1420.

[Kes04] Kassenich, J., Baldwin, D., Rost, R. (2004). *The OpenGL Shading Language.* [WWW] http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf (March 17 2005).

[Kil99] Killgard, M. J. (1999). *Improving Shadows and Reflection via the Stencil Buffer.* [WWW] http://developer.nvidia.com/object/Stencil_Buffer_Tutorial.html (March 17 2005).

[Kil04] Killgard, M. J. ed. (2004). *NVIDIA OpenGL Extension Specifications.* [WWW] http://www.nvidia.com/dev_content/nvopenglspecs/nvOpenGLspecs.pdf (March 17 2005).

[Kle04] Klein, T., Stegmaier, S., Ertl, T. (2004). *Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids*. Proc. of the 12th Pacific Conf. on Computer Graphics and Applications, Seoul, Korea, pp. 186—195.

[Kos04] Koster, M., Haber, J., Seidel, H. (2004). *Real-Time Rendering of Human Hair using Programmable Graphics Hardware.* Proc. of Computer Graphics International, Crete, Greece, pp. 248—256.

[Krü03] Krüger, J., Werstermann, R. (2003). *Linear algebra operators for GPU implementation of numerical algorithms.* ACM Transactions on Graphics, Vol 22, No 3, pp. 908—916.

[Las95] Lastra, A., Molnar, S., Olano, M., Wang, Y. (1995). *Real-time programmable shading.* Proc. of the 1995 Symposium on Interactive 3D graphics, Monterey, USA, pp 59—ff.

[Laz04] Lazányi, I., Szirmay-Kalos, L. (2004). *Speeding up the virtual light sources algorithm.* Proc. of the 20th spring conf. on Computer graphics, Budmerice, Slovakia, pp. 112—120.

[Lef04] Lefohn, A. E., Kniss, J. M., Hansen, C. D., Whitaker, R. T. (2004). *A Streaming Narrow-Band Algorithm: Interactive Computation and Visualization of Level Sets.* Computer Graphics, Vol 10, No 4, pp 422—433.

[Lin01] Lindholm, E., Kilgard, J. M., Moreton, H. (2001). *A User-Programmable Vertex Engine.* Proc. of the 28th conf. on Computer graphics and interactive techniques, Los Angeles, USA, pp. 149—158.

[Mar03] Mark, W. R., Glanville, R. S., Akeley, K., Kilgard, M. J. (2003). *Cg: A system for programming graphics hardware in a C-like language.* ACM Transactions on Graphics, Vol 22, No 3, pp. 896—907.

[McC02] McCool, M. D., Qin, Z., Popa, T. S. (2002). *Shader metaprogramming.* Proc. of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Saarbruecken, Germany, pp. 57—68.

[McC04] McCool, M. D., Toit, S., D., Popa, T. S., Chan, B., Moule, K. (2004). *Shader Algebra.* ACM Transactions on Graphics, Vol 23, No 3, pp. 787—795.

[Mit00] Mitra, T., Chiueh, T. (2000). Three-dimensional computer graphics architecture. Current Science, Vol 78, No 7, pp. 838—846.

[Mor99] Moreton, H. (1999). *High Order Surfaces*. [WWW] http://developer.nvidia.com/object/higher_order_surfaces.html (March 17 2005).

[Mor03] Moreland, K., Angel, E. (2003). *The FFT on a GPU.* Proc. of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, San Diego, USA, pp. 112—119.

[Mus01] Mustafa, N., Koutsofios, E., Krishnan, S., Venkatasubramanian, S. (2001). *Hardware-Assisted View-Dependent Planar Map Simplification.* Proc. of the ACM Symposium on Computational Geometry, Medford, USA, pp. 50—59.

[Nag04] Nagy, Z., Klein, R. (2004). *High-Quality Silhouette Illustration for Texture-Based Volume Rendering.* Proc. of WSCG, Pilsen, Czech Republic, pp. 301—307.

[NVG04] *NVIDIA GPU Programming Guide Version 2.2.0.* (2004). [WWW] http://developer.nvidia.com/object/gpu_programming_guide.html (March 17 2005).

[NVT05] *NVIDIA Technical Brief: SLI Technology.* (2005). [WWW] http://www.nvidia.com/object/IO_18075.html (Match 17 2005).

[Pas04] Pascucci, V. (2004). *Isosurface Computation Made Simple.* Proc. of Symposium on Visualization, Konstanz, Germany, pp. 293—300.

[Pho75] Phong, B. T. (1975). *Illumination for computer generated pictures.* Communication of the ACM, Vol 18, No 6, pp. 311—317.

[Pur02] Purcell, T. J., Buck, I., Mark, W. R., Hanrahan, P. (2002). *Ray tracing on programmable graphics hardware.* ACM Transactions on Graphics, Vol 21, No 3, pp. 703—712.

[Rod04] Rodrigues, R., Fernandes, A. R. (2004). *Accelerated epipolar geometry computation for 3D reconstruction using projective texturing.* Proc. of the 20th spring conf. on Computer graphics, Budmerice, Slovakia, pp. 200—206.

[Seg04] Segal, M., Akeley, K. (2004). *The OpenGL Graphics System: A Specification, Version 2.0.* [WWW] http://www.opengl.org/documentation/specs/version2.0 /glspec20.pdf (March 17 2005).

[Ska04] Skala, V. (2004). *A New Line Clipping Algorithm with Hardware Acceleration.* Proc. of Computer Graphics International, Crete, Greece, pp. 270—273.

[Szi04] Szijártó, G., Koloszár, J. (2004). *Real-time Hardware Accelerated Rendering of Forests at Human Scale.* Proc. of WSCG, Pilsen, Czech Republic, pp. 443—449.

[Vio04] Viola, I., Kanitsar, A., Groller, M. E. (2004). *GPU-based frequency domain volume rendering.* Proc. of the 20th spring conf. on Computer graphics, Budmerice, Slovakia, pp. 55—64.

[Vla01] Vlachos, A., Peters, J., Boyd, C., Mitchell, J. L. (2001). *Curved PN Triangles.* Proc. of the 2001 Symposium on Interactive 3D graphics, North Carolina, USA, pp. 159—166.

[Wan03] Wang, C. L., Yuen, M. F. (2003) *A binary morphology based filtering algorithm for reverse engineering.* International Journal of Advanced Manufacturing Technology, Vol 21, No 4, pp.257—262.

[Wat99] Watt, A. (1999). *3D computer graphics.* 3rd ed. Addison Wesley.

[Wil78] Williams, L. (1978). *Casting curved shadows on curved surfaces.* Proc. of the 5th conf. on Computer graphics and interactive techniques, Atlanta, USA, pp. 270—274.

[Wil83] Williams, L. (1983). *Pyramidal parametrics.* Proc. of the 10th conf. on Computer graphics and interactive techniques, Detroit, USA, pp. 1—11.

[Whi80] Whitted, T. (1980). *An improved illumination model for shaded display.* Communication of ACM, Vol 23, No 6, pp. 343—349.

[Wyl02] Wylie, B., Moreland, K., Fisk, L. A., Crossno, P. (2002). *Tetrahedral projection using vertex shaders.* Proc. of the 2002 IEEE symposium on Volume visualization and graphics, Boston, USA, pp. 7—12.

[Yan04] Yan, J., Kowel, S. T., Cho, H. J., Ahn, C. H., Nordin, G. P., Kulick, J. H. (2004). Autostereoscopic tree-dimensional display based on micromirror array. Applied Optics, Vol 43, No 18, pp. 3686—3696.

[Zac04] Zach, C., Karner, K., Bischof, H. (2004). *Hierarchical Disparity Estimation with Programmable 3D Hardware.* Proc. of WSCG, Pilsen, Czech Republic, pp. 275—282.

[3DL04] *New Wildcat Realizm Graphics Techology*. (2004). [WWW] http://content.3dlabs.com/Datasheets/TechnologyLaunchWhitepaper.pdf (March 17 2005).

Appendix A: GPU Registers

Tables in this appendix contains list of register available for given shader model. These tables are summarization of information contained in the [DX05].

Vertex Shader						
Register	***Description***	***R/W and data type***	***Number of registers for given Shader Model***			
			1.1	***2.0***	***2.x***	***3.0***
v#	Input registers. Contains input provided by the user. Usually, no hardwired input semantics available.	R float4	16	16	16	16
c#	Floating-point constants. Can be set by the application.	R float4	≥ 96	≥ 256	≥ 256	≥ 256
i#	Integer constants. Utilized for static loops. Can be set by the application.	R int4	–	16	16	16
b#	Boolean constants. Utilized for static flow control. Can be set by the application.	R bool	–	16	16	16
s#	Sampler. Utilized for texture access. Can be set by the application.	R sampler	–	–	–	4
r#	Temporary registers. Servers as a local shader memory.	R/W float4	12	12	≥ 12	32
a0	Address register. Utilize as an offset to an array of constants, e.g. c[a0+n].	R/W int4	1	1	1	1
aL	Loop control variable.	R int	–	1	1	1
p0	Predicate register. Utilized for dynamic flow control as a storage place for branching condition.	R/W bool4	–	–	1	1
o#	Output registers. In lower version of shader these registers are available in a form of a register list with denoted semantics, e.g. color register. In such case, some registers may be of float data type instead of float4. Register that contains vertex element has to be set to its homogeneous coordinates.	W float4 float1	2× float 11× float4	2× float 11× float4	2× float 11× float4	12× float4

Pixel Shader						
Register	***Description***	***R/W and data type***	***Number of registers for given Shader Model***			
			1.x/1.4	***2.0***	***2.x***	***3.0***
v#	Input color registers. Contains values provided by the vertex pipeline. For version prior to 3.0 these registers contains only diffuse and specular color.	R float4	2	2	2	10
t#	Input texture registers. Contains texture coordinates provided by the vertex pipeline. For versions prior to 1.4 these registers are filled with samples from texture instead of texture coordinates.	R float4	4/6	8	8	–
vFace	Input face register determining orientation of the face. Only sign is valid. The register can be used only with instructions intended for branching based on comparison.	R float1	–	–	–	1
vPos	Position (x, y) of pixel in screen coordinates.	R float2	–	–	–	1
c#	Floating-point constants. Can be set by the application.	R float4	8	32	32	224
i#	Integer constants. Utilized for static loops. Can be set by the application.	R int4	–	16	16	16
b#	Boolean constants. Utilized for static flow control. Can be set by the application.	R bool	–	16	16	16
s#	Sampler. Utilized for texture access. Can be set by the application. For version 1.4 texture stage number is determined from number of target temporary register.	R sampler	–	16	16	16
r#	Temporary registers. Servers as a local shader memory. For versions prior to 1.4, register r0 serves are an output color register.	R/W float4	2/6	≥ 12 ≤ 32	≥ 12 ≤ 32	32
aL	Loop control variable.	R int	–	–	–	1
p0	Predicate register. Utilized for dynamic flow control as a storage place for branching condition.	R/W bool4	–	–	1	1
oC#	Output color registers. Contains color of pixel. Number of these registers depends on capability of multiple render targets. Thus, the number is either one or four.	W float4	1[49]	1 or 4	1 or 4	1 or 4
oDepth	Output depth register. Can be utilized for modification of depth information associated with pixel.	W float	–	1	1	1

49 For version prior to 2.0, output register functionality is handled by temporary register r0.

Appendix B: GPU Limitations

Maximum number of instructions is limited. Flow control instructions have limitations in its nesting depth. There are several kinds of nesting depth such as static, dynamic, loop, and subroutine call nesting depth. Usually, particular flow control instruction influences particular nesting depth, e.g. a dynamic if-branching modifies only actual dynamic flow control depth.

In addition, static flow control instructions may have limitations in total counts that may appear within a single shader code. Maximum nesting depths as well as total count of instruction is summed in following tables. For more details on specific instructions, refer to [DX05] that is source for tables contained within this appendix.

Vertex Shader				
Value Description	***Value for given Shader Model***			
	1.1	***2.0***	***2.x***	***3.0***
Total maximum number of instruction slots.	128	256	256	≥ 512
Maximum static flow control instructions per shader‡.	–	16	16	∞
Maximum static flow control nesting depth.	–	viz. static flow count‡	viz. static flow count‡	24
Maximum dynamic flow control nesting depth.	–	–	≤ 24	24
Maximum loop/rep flow control nesting depth.	–	1	≥ 1; ≤ 4	4
Maximum subroutine call nesting depth.	–	1	≥ 1; ≤ 4	4

Pixel Shader				
Value Description	***Value for given Shader Model***			
	1.x/1.4	***2.0***	***2.x***	***3.0***
Total maximum number of instruction slots. For lower versions, the instruction count is divided between texture (first number) and arithmetic (second number) instructions.	4 + 8/6 + 8	32 + 64	≥ 96	≥ 512
Maximum static flow control nesting depth.	–	–	24 or 0	24
Maximum dynamic flow control nesting depth.	–	–	≤ 24	24
Maximum loop/rep flow control nesting depth.	–	–	≤ 4	4
Maximum subroutine call nesting depth.	–	–	≤ 4	4