ACCELERATION OF IMAGE PROCESSING ALGORITHMS FOR MEETING SYSTEM

ABSTRACT

For the purpose of Multimodal Meeting Manager, an approach based on omni-directional view system is proposed. The acquired data need to be presented to the human in the appropriate manner, however, not in the "deformed" form obtained through the mirror. The transformation algorithms working with HD resolution are computationally expensive. Therefore specialized hardware such as 3D graphics card is used. The practical application presents the possibility of real-time HD image processing by using common PC hardware.

KEY WORDS

Omni-directional system, 3D acceleration, tracking, image processing, skin color detection.

1. Introduction

The computational load of algorithms for image processing in higher resolution is exceeding the possibilities of today's processors. The Pentium 4 processor running on 3 GHz has theoretical power of 6 GFLOPS and its memory transfers have 5.96 GB/sec in peak. The very strong processing unit in computers, which is often overlooked, is 3D accelerator. Today's common graphical cards as GeForce 6800 Ultra has observed (measured by long shader, consisting of nothing but multiply instructions) power of 40 GFLOPS, with it's memory bandwidth of 35.2 GB/sec in peak. Note GF6800U is not one of top – of – the – line cards, we have 7xxx series today. Graphics hardware computational power is evolving faster than by processors. Moore's law for processors says that performance of processor, developed after single year is 1.5 times higher compared to the older one. Performance of graphics cards doubles every year. The other advantage is scalability of the 3D accelerators to increase the computational power of the whole system. The main aim of our work was to develop real-time system, which is able to process omni-directional video from HD camcorder in resolution 1440×1080 pixels. Among the main tasks belong the omni-directional image transformation into the panoramic or perspective view, parameter estimation for this transformation and pre-processing of the transformed image for tracking purposes as skin color detection etc. The output of this work served as technological demo at CeBit 2006 in Hannover. The following text contains description of the algorithms for omni-directional image transformation, process of 3D hardware programming and technical description of the novel meeting system architecture.

2. Omnidirectional system

In this chapter, we will study the system consisting of an ordinary perspective video camera equipped with a hyperbolic mirror, which allows capturing of a large portion of the space angle - 360×105 degrees field of view.

Fig. 1 Omni-directional meeting recording system

When the image is presented to a human, it needs to be a perspective/panoramic image so as not to appear distorted. A lot of approaches exist to solve this problem depending on the used mirror and camera type.

The simplest transformation of omni-directional image into the panoramic view uses unwrapping of the source image. The necessary parameters are center and radius of the projected circle from the mirror border. The transformation of the output coordinates to coordinates of the captured image can be written as:

$$x_M = CenterX + \cos(\frac{x}{2\pi R_{OUTER}} + Offset) * (R_{INNER} + y)$$
$$y_M = CenterY + \sin(\frac{x}{2\pi R_{OUTER}} + Offset) * (R_{INNER} + y) \quad (1)$$

Offset parameter defines origin of the transformed panoramic image. Parameters R_{OUTER} and R_{INNER} are radii of the outer and inner circle, which define transformation restriction. The last parameters $CenterX$ and $CenterY$ specify the circle center, which is projected from the mirror. The calculated pixels in the camera image do not correspond "one to one" to the pixels of projected image so sub pixel anti-aliasing methods should be used.

Geometry knowledge of the catadioptric system with single effective point enables correct transformation of the mirror image into the suitable form – panoramic, perspective etc. One of the approaches is to use geometrical properties of the mirror for image projection on the cylindrical plane around the mirror axe or perspective plane. Due to the rotational symmetry of the system we only need to know information about the mirror profile. The image formation can be expressed as a composition of coordinate transformations and projections.

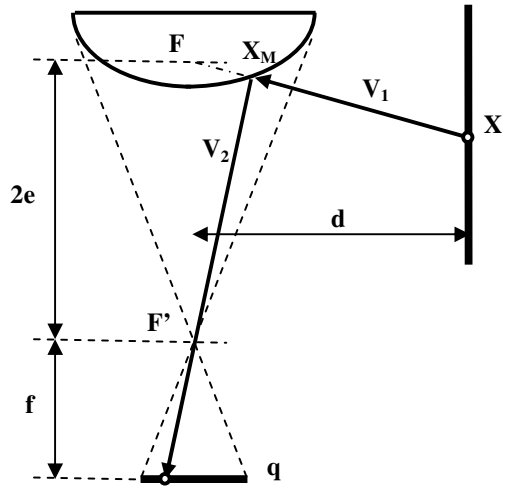

Fig. 2 Imaging model of central panoramic camera with hyperbolic mirror

The center of the coordinate system is chosen in the mirror focal point F. Line v_1 goes through the real world point X on the cylindrical plane and through the focal point F of the mirror. The line equation is $y = qx$. Then, we compute the intersection point X_M on the mirror with the line v_1. The quadratic equation after induction is following:

$$x_M^2 (b^2 q^2 - a^2) - 2x_M qeb^2 + b^2 e^2 - a^2 b^2 = 0 \quad (2)$$

The root x, which presents the x-coordinate of the mirror point, is computed from equation (3). There are two possible solutions, which present two ray intersections with the mirror. We will use the proper one to compute ray reflection.

$$x_{M1,2} = \frac{2qeb \pm 2ab\sqrt{e^2 + q^2 - a^2}}{2(b^2 q^2 - a^2)} \quad (3)$$

This equation is the key in the ray computation. When we know the incoming ray and reflected ray from the mirror, the transformation equations depends on the desired projection. Because the mirror contains the single view point, it is possible to construct geometrically correct perspective view from this point.

3. Programmable part of OpenGL pipeline

Classical (or more precisely ‚fixed function') OpenGL pipeline did not programmer allow to do very much. Basically, it enabled drawing textured, lit and fogged primitives. Recent graphics hardware introduces so called programs or shaders, which are used to bypass certain parts of fixed function pipeline. In fact we can bypass two parts of pipeline – vertex processing (so we're talking about vertex program / vertex shader) or fragment (pixel) processing (fragment program / fragment shader). Note program and shader differs in used language only, but the functionality is basically the same. We will prefer word program from now. The basic structure of OpenGL pipeline looks like:

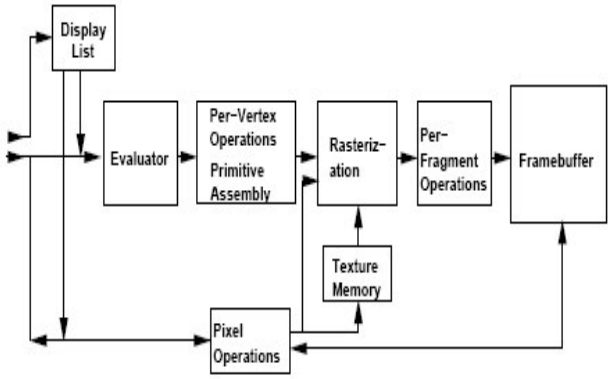

Fig. 3 Block diagram of OpenGL

Vertex programs allow us to replace per-vertex operations, namely all kinds of transformation (vertex, normal, texture coordinate), texture and fog coordinate generation, lighting and color material application. It does not sound very impressive, but you do not need - for example - to use texture coordinates for texture lookup later on fragment processing stage. You can write fragment program as well and use them as some data input.

The following operations are not allowed in vertex program: perspective division, view port mapping, primitive assembly, clipping, backface culling, two-sided lighting selection, polygon mode processing, depth range nor polygon offset.

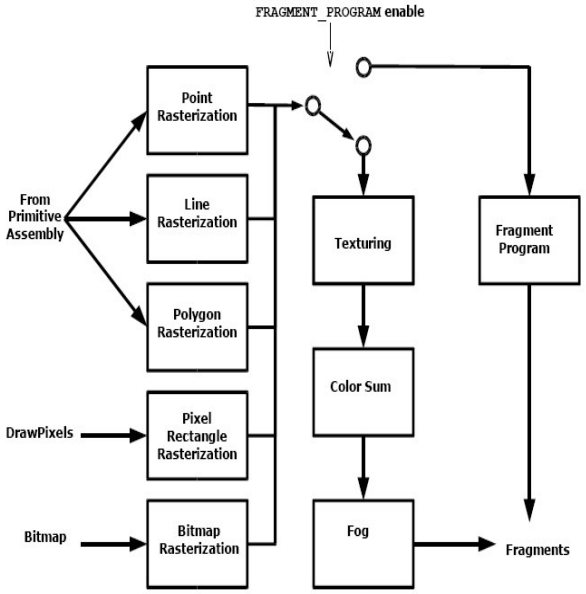

Fig. 4 OpenGL fragment processing pipeline

As you can see in fragment program, we get interpolated values from vertex program (fragment is a single pixel only, not the whole polygon scanline) or values provided by fixed function pipeline in case vertex program is not enabled. We can take those values and use them to determine output color. We can output more values or discard the fragment so nothing is written into the output buffer(s).

4. Basic scheme of general-purpose computation in OpenGL

Data provided to shaders can exist in several forms. Those are textures, vertices, program variables and OpenGL state information. OpenGL offers us:

- color texture border
- bilinear and trilinear filtering for free (OpenGL texture coordinates are floating point numbers, bottom left corner has coordinates [0, 0], upper right corner [1, 1])
- mip-mapping, automatical level of detail selection, based on spatial sampling density
- texture repeat modes (determine what to do when texture coordinate is below 0 or above 1), modes are repeated, mirrored repeated, clamp, clamp to edge, clamp to border
- anisotropic texture filtering, levels of anisotropy up to 16

In OpenGL offers us various kinds of textures:

- One-dimensional textures – basically a long strip of values accesed by a single coordinate (in fact it‘s just a special case of 2D texture and it has the same limitations)
- Two-dimensional textures – classical images, but with a few limitations. There‘s maximal texture size limit (usually 2048×2048 or 4096×4096 pixels), texture size is limited to powers of two as well, but there are some extensions that enables use of non-power of two sized textures.
- Three-dimensional textures – used for volumetric data. Again, there is size limitation (512×512×512 on Nvidia cards, 4096×4096×4096 on ATI cards) and dimensions are limited to power of two.
- Cube-map textures – special kind of texture. Basically it is a cube, assembled of six square 2D textures. Any 3d vector can be used for texture lookup (any means even un-normalized, so cubemaps can contain precomputed unit vector values and therefore be used for vector normalisation (so called normalisation cubemap); there‘s normalize instruction in shaders, but in some cases texture lookup is cheaper) There are basically the same limits as for 2D textures with adition that textures must be square (width equal to height).

Texture pixel formats allow one, two three and four-component textures, while all components must be of the same data type. Data types are signed or unsigned integers of widths 2, 4, 5, 8, 10, 12 and 16 bits or (signed) floating-point values with 16 bits (1 bit sign, 5 bit exponent and 16 bit mantissa) or 32 bits. Textures can be accessed in fragment programs and with certain limitation to texture format (floating-point textures only) and texture fetch speed even in vertex programs.

OpenGL vertices are 2, 3 or 4-dimensional position (fourth dimension is w, for most vertices will be 1, you can move vertex into infinity by setting w to 0) accompanied (optionally) by:

- vertex normal (3 component vector)
- color (4 componens)
- secondary color (3 components)
- texture coordinates (4 component vectors, on most hardware there‘s 8 of them)
- fog coordinate (single value).

You can keep vertices in system memory (which is slower) or in graphics card memory (which require some extensions, but is significantly faster). Having vertices in graphics card memory offers possibility to copy texture data into vertex buffer or vice-versa (again, certain extensions are necessary). Programs can define set of variables (more on variable types later) that can be set before calling program. It means they can not be changed during the rasterization of a single primitive, which is purpose of vertex properties. Programs can read OpenGL state variables, such as transformation matrices, material and light properties, point properties, depth range, texture environment and generation parameters as well as to fog parameters. Specific limits such as maximal number of lights, clip planes, texture units, texture coordinates and draw buffers are made available trough constants.

Classical OpenGL data output is frame buffer, visible trough some window. This has some limitations, such as maximal window size, pixel format (pixel format limitations arise from need to be able to display frame buffer contents) and sometimes we simply don‘t want user to see what is being computed.

As response to problems mentioned in paragraph above, two extensions supporting rendering to off-screen buffer originated. Namely, they are pixel-buffer (or P-buffer) and frame-buffer object (FBO). Offscreen buffer size is limited only by maximal size. Pixel formats are implementation specific, but basically you should be able to set the same pixel formats as for textures.

Pixel buffers are older and more widely supported extensions, but it has a big disadvantage – they are not cross-platform extension. There are two similar extensions for both linux and windows, but you need to write two code paths.

On the other hand, FBO‘s are pure OpenGL extension, which means that they are platform independent extension.

There was already mentioned rendering to several outputs simultaneously. It is possible trough extension GL_ARB_draw_buffers and we can have one, two, four, eight or sixteen draw buffersv (we have an array of 16 values we can write to in fragment program so we can not bind different fragment program nor set different OpenGL state for separate draw buffers).

Sometimes, we will need to process data in multiple steps (for example, we could want to outputs form more steps, or simply because implementation of certain algorithms is simpler this way). We need to somehow copy data from our rendering buffer to either texture or to vertex buffer (rendering to OpenGL state is impossible).

FBO‘s have native direct rendering to texture capability, which means instead of allocating data for frame buffer. We specify classical OpenGL texture which is going to be written to. Pixel-buffer objects can render to texture trough extension. We have to bind multiple textures to render for multiple draw buffers (they need to have the same dimensions). Up to now, the rendering to 2D texture was described. Note that 1D texture is just a case of 2D so we can render to 1D texture as well. Rendering to cube-map or 3D texture is done in multiple passes as rendering to cube sides or volume slices respectively.

Direct rendering to vertex array is possible trough GL_ATI_superbuffer, which is nowadays supported on ATI cards only. The other cards enable rendering to texture and copy its data to vertex buffer.

5. Performance and implementation description

Transformation shaders used in the meeting system can be divided into two categories. The first one is group of fragment shaders, taking texture with mirror image as source and directly drawing transformed result. The fullscreen quad is drawn with proper texture coordinates, which are used as input for calculations. This way of calculation is not very effective, because of the computational redundancy. The distance from the center of the mirror is function of y-coordinate in resulting image, whereas the angle of half-axis is function of corresponding x-coordinate, so we theoretically only need to transform all x-coordinates and all y-coordinates to get the very same results, but that would require some additional textures and context switches. That is why we decided for such kind of simpler solution.

Shaders that calculate transformed texture coordinates in vertex shader (i.e. not for every pixel on-screen) and are linearly interpolated fall into second category. Of course, the transformations we are working with are everything but linear so we can not get away with fullscreen quad any longer. We need quite a good subdivision for our fullscreen quads. Our implementation uses vertex buffer object to store big quad strip vertices (in fast memory of the graphics card) which we draw instead of fullscreen quad. It is not ideal solution, but as an advantage we can use anisotropic filtering, provided by the hardware. Anisotropic filter is taking sampling density in account which results into elliptic samples (multiple points are sampled) rather than circular ones. This was considered quite a good solution, because with quad tesselation to 128×128 sub-quads and perspective correct projection,

the whole transformation does not take much more than 1.5 milliseconds (see the table Tab.1).

Transformation	Additional image processing		
	none	grey	Skin det.
Simple	4.3ms	4.4ms	9.2ms
Geometric	5.8ms	5.9ms	8.9ms
Tesselation 128×128	2.0ms	2.0ms	5.3ms
Tessellation 128×128, Perspective correction	1.6ms	1.7ms	5.1ms

Tab. 1 Speed tests for various image transformations

When looking back, it is quite a lot of time saved so we are reconsidering transformations of x and y coordinates separately. In the end, we can cache results in texture until parameters of transformation change.

Set of tests were performed to show the computational power of 3D accelerators. In the beginning we feared of too long time spent on image transfers from and to graphics card, but we implemented asynchronous transfers which are effective. Along with the transformation, shaders can precalculate some data for image recognition stage (skin colour tracking). The results we get allows us to perform better texture filtering (like cubic spline filter which we are currently implementing) we couldn't afford on CPU.

6. Experimental system

A demonstration application for so-called mobile meeting room was developed for an evaluation of proposed approaches. The application processes a high resolution video stream acquired through an omnidirectional system. The particular frames are transformed with unwrapping algorithm into the panoramic view and the human detection and tracking algorithms are applied on this picture for highlighting of the meeting participants. Additionally, speech processing is used for identification of active speaker and keyword detection. The application have to work in real-time, so efficient implementation is the key issue. The structure of the implemented application is presented in Fig. 5.

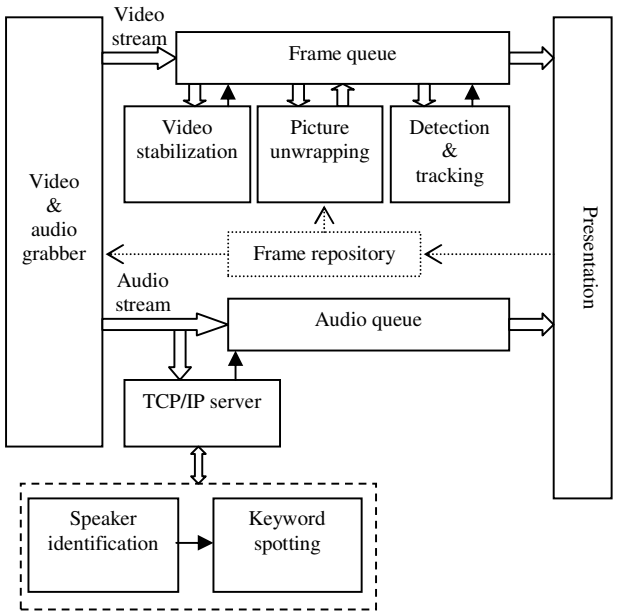

Fig. 5 Internal structure

Source video and audio streams are acquired through a grabber based on Microsoft DirectShow technology [7]. Input signals from camera and microphones are processed through a graph containing some filters for demultiplexing of MPEG2 streams that is provided by camera, decompression of video frames, and grabbers for audio/video frames. In addition, audio streams are passed through delay filters to players. Implemented graph is presented in.

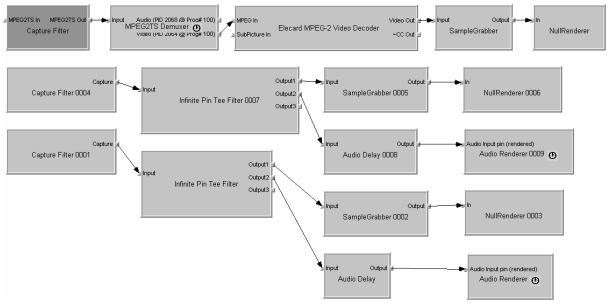

Fig. 6 DirectShow graph

Grabbed video frames are together with timestamps stored into the frame queue. Following blocks can work on their sets of data from the certain time point. The main effort was to all blocks work in parallel, so the multi-core CPU can be efficiently used. The queue is also necessary for synchronization of audio/video streams during presentation of the results. At first, a stabilization of the given frame is evaluated, in fact the centre of omnidirectional mirror is found. Further, picture unwrapping algorithm is applied and the panoramic view is stored into the queue. Finally, meeting participants are detected and tracked using [2]. The results of this detection are also stored into the queue with current

timestamp. In addition, speech processing part evaluate active speaker and checks some keywords. These two blocks are running on a secondary computer connected via TCP/IP server, because as experiments shown the image processing utilizes whole CPU on main computer. Distribution of video processing across several computers is unacceptable, because data flow of transferred frames would be too big. However, a multiprocessor computer can be used as well.

Processing of high resolution data is quite demanding, the data rate of source video stream in 1440×1080 pixels is approximately 148 MB/s (4 byte per pixel). The memory transfers of source frames have to be minimized, because every memory transfer of the picture consume some time. Therefore, source frames are stored only once in the frame queue and the references are used instead of coping of the data. Also, if the panoramic view is generated, it is stored into the queue, and the references to parts of this picture are used in "Detection & tracking" block and "Presentation" block. In addition, the memory allocation of such big images is also problematic. Therefore, the own memory manager was implemented. Pictures, which have been already presented, are not freed but stored into the "Frame repository" and grabber then uses these frames instead of new allocated frames.

The application is being executed on a computer with AMD Athlon 64 X2 Dual 4400+ CPU with 2 GB of RAM. NVIDIA GeForce 6600 graphic card is used for unwrapping of omnidirectional pictures, HDV Sony camera HDR-FX1E acquires picture from hyperbolic mirror, and professional sound card RME Fireface 800 records four sound channels for speech processing. Together with the source video stream in HDV resolution is in the application displayed (see Fig. 7) whole unwrapped picture in 1024×324 resolution and four views in resolution 204×324 pixels, one for each meeting participant. The application is able to process up to 25 frames per second(FPS) at this configuration. The same computation, but only on the CPU, has about 1.6 FPS.

Fig. 7 Application interface

7. Conclusion

The goal of this contribution was to present a process how to accelerate algorithms for meeting system and practical application. It has been shown that the 3D graphics card can significantly speed up the whole computation process. The proposed approaches were verified in experimental system for mobile meeting room. The system, where the graphic card was used, is about 20 times faster than computation only on CPU. The significant part of computation was processing of the HD video.

Acknowledgements

This work was partly supported by the European Union 6th FWP IST Integrated Project AMI (Augmented Multi-party Interaction, FP6-506811, publication).

References

[1] I., Potúček, Automatic Image Stabilization for Omni-Directional Systems, In: Proceedings of the Fifth IASTED International Conference on VISUALIZATION, IMAGING,AND IMAGE PROCESSING, Benidorm, ES, ACTA Press, 2005, s. 338-342.

[2] M. Hradis, R. Juranek, Real-time Tracking of Participants in Meeting Video, Proceedings of CESCG, Wien, 2006.

[3]] S., Sumec, I., Potúček, P., Zemčík, AUTOMATIC MOBILE MEETING ROOM, In: Proceedings of 3IA'2005 International Conference in Computer Graphics and Artificial Intelligence, Limoges, FR, 2005, s. 171-177, ISBN 2-914256-07-8.

[4] M., Segal, K., Akeley, Ch., Frazier, J., Leech, P., Brown, The OpenGL(R) Graphics System, A Specification, Version 2.0, 2004

[5] I., Elhassan (NVIDIA Corp), Technical brief - Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL, s.1-5, 2005.

[6] European research AMI - Augmented multi-party interaction, http://www.amiproject.org/

[7] Microsoft DirectX Developer Manual, http://msdn.microsoft.com/directx/