GPU Based Evaluation and LOD Rendering of NURBS Surfaces

Tan Dunming[1, 2], Zhao Gang[1] and Lu Hu[3]

[1]State Key Laboratory of Virtual Reality Technology and Systems, Beihang University, zhaog@buaa.edu.cn
[2]The First Aeronautical Institute of PLA Air Force, tan_dunming@foxmail.com
[3]Shanghai Aircraft Manufacturing CO., Ltd, luhu@comac.cc

ABSTRACT

This paper proposes a new GPU method for the evaluation and LOD (Level of Detail) rendering of NURBS surfaces. Compared with the existing evaluation method of basis function, the approach simplifies the process by just one-pass and saves the graphics memory with only one texture array. The wavelet based LOD rendering method also saves the time to re-evaluate and re-transfer the data for rendering.

Keywords: GPU, NURBS, De Boor's algorithm, LOD, wavelet.
DOI: 10.3722/cadaps.2012.707-719

1 INTRODUCTION

1.1 Background and Problem

Non-Uniform Rational B-Spline (NURBS), a mathematical model commonly used in Computer Aided Geometry Design (CAGD), is the foundation of Computer Aided Design (CAD) and Computer Aided Manufacturing (CAM) for representing curves and surfaces. NURBS is also incorporated into international standards of Initial Exchange Specification (IGES), as well as Standard for the Exchange of Product model data (STEP) [1].

The evaluation and rendering of NURBS surfaces are very time consuming and there is an urgent need for fast evaluation and rendering methods. For example, Boeing 737, the first aircraft designed by CAD and represented by NURBS surfaces, was evaluated and tessellated into 350M triangles [2], which is a greate challenge for both the evaluation and rendering.

Recently, there are some researches that use GPU to accelerate the evaluation and rendering of NURBS surfaces. But those methods use the basis functions to evaluate, which make the computing in multi-pass and consume extra graphics memory. Besides, those methods still use traditional methods for Level of Detail (LOD) rendering, which spend extra time to re-generate and re-transfer the data.

1.2 Our Work

In this paper, the authors present a new method to evaluate and render the NURBS surfaces by adopting GPU shader model 4.0 [3]. The evaluation is implemented with de Boor's algorithm by GPU fragment shader. To avoid the bottle-neck of data transfer from GPU to CPU, the evaluated data are kept in the graphics memory. The rendering is accomplished with the assistance of GPU vertex texture fetch, which uses the graphics texture array for vertex position and no re-transfer of the evaluated data from GPU to CPU. Wavelet multi-resolution analysis is used to accelerate the LOD rendering, which can avoid the time to re-evaluate NURBS surfaces with different resolutions.

1.3 Paper Overview

The remainder of the paper is organized as follows: Section 2 starts with a brief overview of some existing GPU based methods for evaluating and LOD rendering of NURBS surfaces. Section 3 provides some insights into basics of NURBS, the de Boor's algorithm, GPU texture representation of NURBS surfaces, the computation mode of GPU fragment shader, the GPU evaluation and rendering system developed. Section 4 describes the GPU and LOD rendering of NURBS surfaces. Section 5 gives the results and analysis. Section 6 discusses the conclusions.

2 RELATED WORK

Nowdays, GPU has evolved into an extremely flexible and powerful processor with high performance in parallel computing. GPU develops rapidly and is now used in many fields of science and technology including the evaluation and rendering of NURBS surfaces in CAGD.

2.1 GPU Based Evaluation of NURBS Surfaces

GPU for per-pixel evaluation of parametric surfaces was first used by Kanai and Yasui [4] in order to achieve high quality surface rendering. In their algorithm basis function coefficients were provided by a fragment shader and multiplied by control points to evaluate Bezier and B-spline surfaces. However the algorithm is difficult to use if the orders of surfaces are not fixed. Furthermore, it simplifies non-linear parametric NURBS surfaces as linear ones and the algorithm would be too slow for real-time rendering if the number of surfaces exceeds ten.

Bezier patch was used by GPU vertex shader to approximate NURBS [5], However, it needs to convert a NURBS surface into multiple bi-cubic Bezier patches by CPU first. Meanwhile, the method is not suitable to approximate high order NURBS surfaces with accuracy.

Basis function was used by GPU fragment shader to evaluate NURBS surfaces by multiplying the basis functions with the control points [6]. However, different GPU fragment shader must be used for NURBS surfaces of different orders and therefore not suitable for general purposes.

Then GPU was used to evaluate NURBS surfaces of arbitrary order [7,8]. Packing of the knots and basis function data was optimized to reduce the data to be transferred. But CPU was still used to find the u and v knot spans, though the basis function was computed on the GPU. Both of the two methods require computing basis function in multi-pass and spend extra graphics memory to store basis function of different orders.

2.2 GPU Based LOD Rendering of NURBS Surfaces

LOD is important for rendering especially when applied to massive models. LOD can be divided into discrete LOD and continuous LOD [9]. Discrete LOD may cause geometry popping when changing the LOD level and spend extra memory. Continuous LOD is based on the theory of Hoppe's Progressive Meshes [10] to solve geometry popping. However, extra time is needed when changing the LOD level.

There are some methods of GPU based LOD rendering of NURBS surfaces.

Rockwood gives a method by first converting all surfaces to Bezier patches and then tessellating into grids of rectangles with different resolutions [11]. Guthe uses CPU to select sufficiently accurate rational bi-cubic Bezier patches for GPU to render [5], only the NURBS connectivity index data are generated and transferred to evaluate and render NURBS surfaces.

Krishnamurthy used different parametric resolution for discrete LOD, Vertex Buffer Object (VBO) is used to avoid transferring the evaluated data back from GPU to CPU for rendering [7,8]. However, the method requires re-evaluating NURBS surface with different parametric resolutions and re-generating the connectivity index data by CPU. Then the evaluated NURBS surfaces are transferred to GPU. Therefore, this approach is not efficient enough for rendering compared with hardware accelerated display list.

Han used GPU geometry shader to avoid suffering from limited bandwidth of transferring data to GPU [12], the connectivity index data of different LOD are generated by GPU geometry shader, so there is no CPU overload. However, due to the limitation of current GPU geometry shader, the number of emits primitives is limited. So the parametric resolution of the NURBS surface cannot achieve high accuracy.

3 GPU BASED EVALUATION OF NURBS SURFACES

3.1 Basic NURBS Theory

NURBS curves and surfaces are defined by control points, weight, knot vector and order. NURBS curve is a special case of NURBS surface, which only has them in one direction. For simplicity, we first introduce NURBS curve and then NURBS surface.

The mathematical definition of NURBS curve is shown in Eqn. (1). $C_{(u)}$ is the NURBS curve and u stands for the interpolation parameter, the curve is influenced by n control points P_i, each control point corresponds to a weight w_i, the curve has an order of k and knot vector U_i shown in Eqn. (2). Eqn. (3) and Eqn. (4) are the recursive basis functions definition.

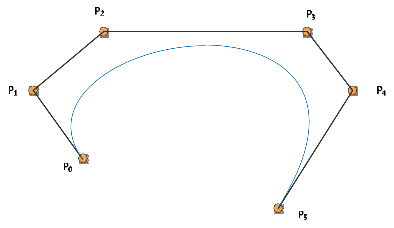


Fig. 1: NURBS curve with control points.

$$C(u)=\frac{\sum_{i=0}^{n} w_i P_i N_{i,k}(u)}{\sum_{i=0}^{n} w_i N_{i,k}(u)} \tag{1}$$

$$U_i = [u_0, u_1, ..., u_{n+k+1}] \tag{2}$$

$$N_{i,k}(u) = \frac{u-u_i}{u_{i+k}-u_i} N_{i,k-1}(u) - \frac{u_{i+k+1}-u}{u_{i+k+1}-u_{i+1}} N_{i+1,k-1}(u) \tag{3}$$

$$N_{i,0}(u) = \{ \begin{matrix} 0 & u_i \le u < u_{i+1} \\ 1 & otherwise \end{matrix} \tag{4}$$

As shown in the formulas above, NURBS curve is evaluated by basis function in recursion. As far as we know, all GPU NURBS algorithms have been implemented in this way. However, it is time and memory consuming and thus we can conclude that this is not the best way.

3.2 De Boor's Algorithm

The De Boor's algorithm is fast and stable for evaluating NURBS curves and surfaces [13]. It is a generalization of De Casteljau's algorithm and is shown in Eqn. (5) to Eqn. (7).

$$C(u^*) = \sum_{j=0}^{n} w_j P_j N_{i,k}(u^*) = \sum_{j=i-k}^{i-1} w_j P_j^l N_{j,k-1}(u^*) = P_{i-k}^k \tag{5}$$

$$P_j^l = \{ \begin{matrix} P_j & l=0 \\ (1-\alpha_j^l) P_j^{l-1} + \alpha_j^l P_{j+1}^{l-1} \end{matrix} \tag{6}$$

$$\alpha_j^l = \frac{u-u_{j+1}}{u_{j+k+1}-u_{j+1}} \tag{7}$$

Compared with basis function method which needs all control points multiplied by basis function and evaluated in multi-pass, De Boor's algorithm only needs *k+1* control points of P_j *(j=i-k,i-k+1,…, i)* and evaluated without recursion. Meanwhile, it avoids computing the basis function, which saves time and memory.

However, up to now this algorithm has not carried out by GPU parallel computing.

3.3 GPU NURBS Evaluation by De Boor's Algorithm

We choose OpenGL Shading Language (GLSL) to implement our GPU NURBS De Boor's algorithm for its cross-platforms compatibility. Besides, this is easy to port to other GPU language such as High Level Shading Language (HLSL) of Direct3D or CG.

3.3.1 Prepare GPU NURBS texture array data

The first step is to prepare the NURBS data for GPU to evaluate. For GPU computing, the native data layout is a two dimensional array [14] and is called texture in graphics. So, in order to evaluate NURBS by GPU, we need to prepare the NURBS data and then transfer it to GPU.

In our algorithm only control points P_i with weight w_i and the parametric result *C(u)* are stored in texture array. *GL_TEXTURE_RECTANGLE_ARB* is used to avoid the restriction of texture size to be power of 2 and we choose *GL_RGBA32F_ARB* texture format because each control point has *x,y,z* coordinates and weight *w*, corresponding to the *RGBA* texture mode. So each control point corresponds to a texture pixel. One or more NURBS curves with the same oder, control points number and knot vector can be evaluated at the same time, which is useful for NURBS surfaces.

So the data layouts of the NURBS control points with weights are somewhat like Fig. 2. The *RGBA* components respond to *x, y, z, w*. Control points of each NURBS curve may be stored in one direction while multiple NURBS curves, which have the same oder, control points number and knot vector, may be stored in the other direction.

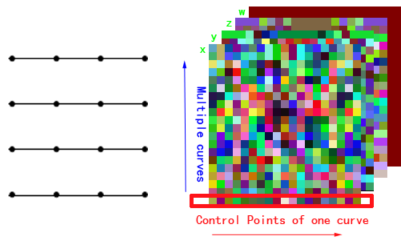


Fig. 2: Control points texture data.

After preparing the NURBS texture array data, it is transferred from memory to GPU by *glTexImage2D* in order to evaluate NURBS with GPU shader.

Order k and knot vector U_i are not stored in texture because they use less memory and are transferred to GPU directly as uniform variables for cache-coherent in order to reduce data access time, see 3.3.2 for detail.

3.3.2 Evaluate NURBS curves

In GPU evaluation, there exist several input texture array data, a computation kernel and several output texture array data. The computation kernel is a fragment shader of GLSL which is executed by parallel computing of GPU. It takes input from input texture array data and writes results to FBO (Frame Buffer Object) rendering target of output texture array data. GPU evaluation is executed by drawing, because fragment shader is executed when a pixel is drawn and the coordinate of the pixel can be used as NURBS parameters of u.

First, uniform variable n representing the number of controls points, the order k, the parametric resolution and direction of the control points are transferred to GPU.

We draw serials of quads corresponding to the effective knot vector span region from u_k to u_n, with the width equals to parametric resolution and height equals to number of curves. The span region index i (described in 3.2), the related knot vector u_{i-k}, u_{i+k+1} (see Fig. 3), which are the same in one quad, were transferred to GPU as uniform variables directly to achieve cache-coherent. Compared with other GPU NURBS algorithms, we avoid the time to find the knot span region index i in which u lies by drawing knot vector span region quads and tell GPU directly.

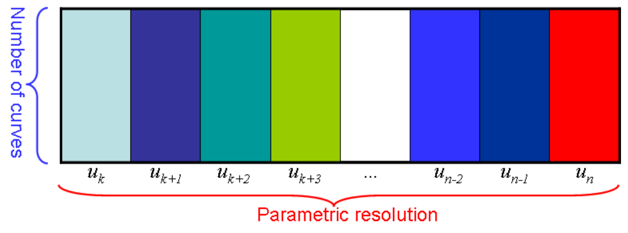


Fig. 3: Draw knot span region with related knots vector for NURBS GPU evaluation

Our GPU kernel of De Boor's algorithm is quite simple and executes all evaluation on GPU. The steps will now be described consecutively.

- Compute the u value corresponding to the drawn pixel. In our NURBS evaluation, we scale the range of effect u form 0 to 1, and we use GL_TEXTURE_RECTANGLE_ARB texture format with texture coordinates range from 0 to texture size, so the u value equals to the texture coordinate of that pixel divided by parametric resolution, described in Fig. 3.

- Get the control points of Pj (j=i-k,i-k+1,**é**, i) by GPU texture sampler from the control points texture prepared in 3.3.1 and shown in Fig. 2.
- Execute De Boor's algorithm described in 3.2 on GPU with shader model 4.0, which is available in almost every current GPU.

The evaluation result is writing to FBO rendering target texture, namely the NURBS curve texture array data for further rendering.

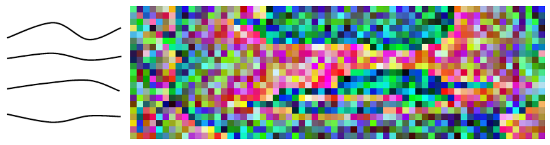


Fig. 4: *C(u)* texture data of evaluated NURBS curves.

3.3.3 Evaluate NURBS surfaces

NURBS surfaceÕs definition is shown in Eqn. (8).

$$S(u,v)=\frac{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{i,j}P_{i,j}N_{i,k}(u)N_{j,l}(v)}{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{i,j}N_{i,k}(u)N_{j,l}(v)} \tag{8}$$

Evaluation of NURBS surfaces is similar to NURBS curves. There is one more evaluation in *v* direction. First, NURBS surface control points are evaluated as NURBS curves in one direction described in 3.3.2. Then the evaluated points between *C(u)* curves are treated as control points again and evaluated in the other direction and finally get the evaluated NURBS surface mesh. See Fig. 5 for how NURBS surface is evaluated by first evaluating NURBS curve in *u* direction.

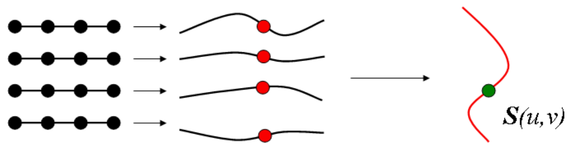


Fig. 5: NURBS surface evaluation.

When evaluating from *C(u)* to *S(u,v)*, the GPU evaluation kernel is the same, however the drawing of knot span region quads is changed to the other direction, see Fig. 6. The uniform variable of NURBS evaluation shader representing the direction of control points is also changed.

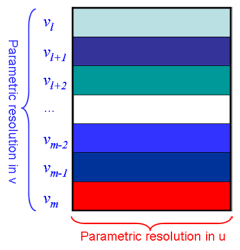


Fig. 6: Knot span region in the other direction.

We do not need to prepare the NURBS surface control points texture data and transfer them into graphics memory, because the curve evaluation texture array data *C(u)* are used as NURBS surface control points texture array data directly. The NURBS surface evaluation result is written to surface evaluation texture array data.

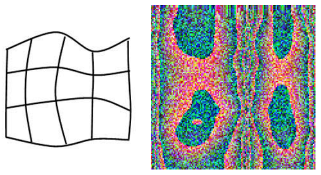


Fig. 7: *S(u,v)* texture array data of NURBS surface.

4 GPU BASED RENDERING OF NURBS SURFACES

We will first introduce NURBS rendering methods, especially the GPU rendering method, the the wavelet multi-resolution analyze based GPU LOD algorithm is then introduced.

4.1 Traditional Rendering

Traditional rendering is implemented as follows: First, read the GPU evaluation results of texture array data from FBO and write them in the memory. Second, use OpenGL to render triangles or quads of the evaluated meshes and send the rendering data to the graphics memory again for rendering. Because exchange data between memory and GPU is very slow [5], there will be little advantage compared with CPU NURBS evaluation algorithm. We do not suggest although it is quite simple.

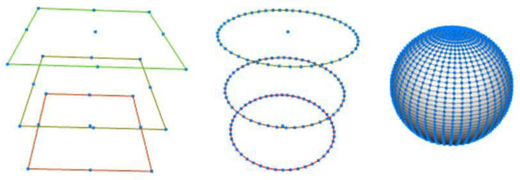


Fig. 8: Rendering of control points, curves and surface of a sphere.

4.2 GPU NURBS Rendering

NURBS rendering can also be accelerated by GPU. With the fast development of the GPUís program abilities, float texture array data can be accessed by either vertex shader or geometry shader ever since GPU shader model 4.0 [3]. So we can avoid the transferring of evaluated curve or surface texture array data back to memory and then re-transferring rendering data to GPU again. This is achieved by first rendering the geometry topology connectivity index of the NURBS curves or surfaces, while in GPU vertex shader the topology connectivity index is converted into real evaluated curve or surface coordinates by looking up the evaluated texture array data. In this way, GPU rendering speed is much faster than traditional method.

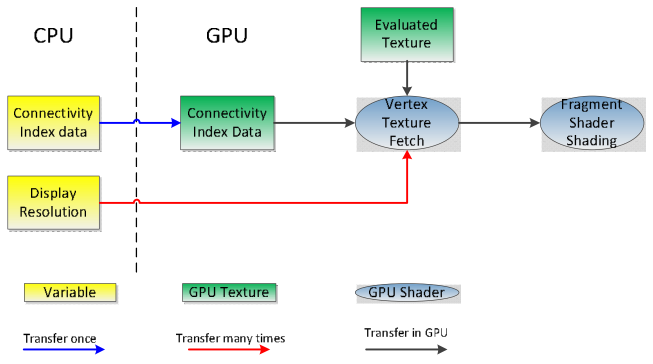


Fig. 9: GPU NURBS rendering work flow.

Fig. 9 shows the rendering work flow. While rendering, the mesh connectivity index data are transferred to GPU only once and compiled to display list for fast rendering, Vertex texture fetch is used to convert display list of connectivity index data to evaluated NURBS curves or surfaces coordinate data. GPU fragment shader is used for lighting and outputs the final NURBS surface. So the GPU NURBS evaluation and rendering are separated at different stages and do not affect each other.

The pseudo code below describes the GPU NURBS rendering, the connectivity index data are transferred to GPU by OpenGL rendering calls and stored in gl_PositionIn. With texture2Drect, the connectivity index data are converted to the real location of the evaluated $S(u,v)$, then three transformed vertexes are emitted to generate a triangle.

```
é
uniform sampler2DRect SuvTexArray;

é
vec2 TexCoord = gl_PositionIn[0].xy;
vec4 Suv = texture2DRect(SuvTexArray, TexCoord.xy);
gl_Position = gl_ModelViewProjectionMatrix* Suv;
EmitVertex();
é

EndPrimitive();//
é
```

4.3 GPU NURBS LOD Rendering

LOD is an important technology in real-time rendering. We implement LOD of NURBS by wavelet. Wavelet has special advantage in multi-resolution analyze and GPU accelerated wavelets analysis is used for LOD rendering.

4.3.1 Wavelet basis

Wavelet is a new concept in multi-resolution due to its major advantages, namely it can keep the main characteristic of the targeting object as well as its detail characteristic. This is done by wavelet transform and reverse transform.

Haar wavelet, which is the most simple and fast wavelet, is select to perform progressive LOD coarse and refinement. The scaling function is defined by Eqn. (9) and the wavelet function is defined by Eqn. (10).

$$\phi(t) = \{ \begin{matrix} 1 & 0 \le t < 1 \\ 0 & otherwise \end{matrix} , \qquad \phi_{j,k}(t) = \phi(2^j t - k) \quad k = 0, 1, ...2^j - 1 \tag{9}$$

$$\varphi(t) = \{ \begin{matrix} 1 & 0 \le t < 0.5 \\ -1 & 0.5 \le t < 1 \end{matrix} \qquad \varphi_{j,k}(t) = \varphi(2^j t - k) \quad k = 0, 1, ...2^j - 1 \tag{10}$$

Suppose there is a discrete sequence of A_n composed by $\{a_{n,0}$, é, $a_{n,2^n-1}\}$, which can be represented by Eqn. (11). So A_n is transferred by Haar wavelet into the main part of A_{n-1} composed by $\{a_{n-1,0}$, é, $a_{n-1,2^{n-1}-1}\}$ and the detailed part of D_{n-1} composed by $\{d_{n-1,0}$, é, $d_{n-1,2^{n-1}-1}\}$.

$$a_{n,0}\phi_{n,0}(t) + ...a_{n,2^n-1}\phi_{n,2^n-1}(t) = a_{n-1,0}\phi_{n-1,0}(t) + ...a_{n-1,2^{n-1}-1}\phi_{n-1,2^{n-1}-1}(t) + d_{n-1,0}\varphi_{n-1,0}(t) + ...d_{n-1,2^{n-1}-1}\varphi_{n-1,2^{n-1}-1}(t) \tag{11}$$

Two dimensional Haar wavelets have corresponding scaling function, wavelet function and transform functions shown in Eqn. (12) and Eqn. (13).

$$\begin{aligned} \psi(x, y) &= \phi(x)\phi(y) \\ \psi^1(x, y) &= \phi(x)\psi(y) \\ \psi^2(x, y) &= \psi(x)\phi(y) \\ \psi^3(x, y) &= \psi(x)\psi(y) \end{aligned} \tag{12}$$

$$\sum a_{k,m}^{j+1}\varphi_{j+1,k,m} = \sum a_{k,m}^{j}\varphi_{j,k,m} + ...\sum d_{k,m}^{j,1}\psi^1_{j,k,m} + \sum a_{k,m}^{j,2}\psi^2_{j,k,m} + ...\sum d_{k,m}^{j,3}\psi^3_{j,k,m} \tag{13}$$

4.3.2 Wavelet of LOD for NURBS

When applying wavelet transform to multi-resolution analysis of NURBS, for NURBS curves, it is one dimensional wavelet transform executed in the row direction shown in Fig. 10 (a). For NURBS surfaces, it is a two dimensional transform that first executed in the row and then in the column direction shown in Fig. 10 (b).

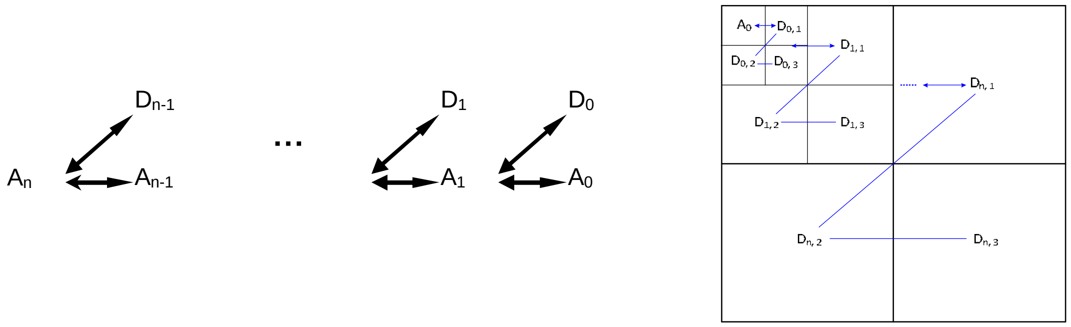


Fig. 10: Wavelet transform and reverse transform: (a) one dimensional transform, and (b) two dimensional transform.

The GPU accelerated wavelet multi-resolution analysis is shown in Fig. 11. It takes the evaluated texture array data as A_n, which are the inputs of GPU wavelet transform shader. The Wavelet transform is executed by GPU shader. It writes the transformed A_{n-1} and D_{n-1} to the FBO rendering targets of next LOD level of texture array data. The evaluated texture array data of A_n can be transformed multi-times until A_0. Meanwhile, this process can be reversed with similar reverse transform.

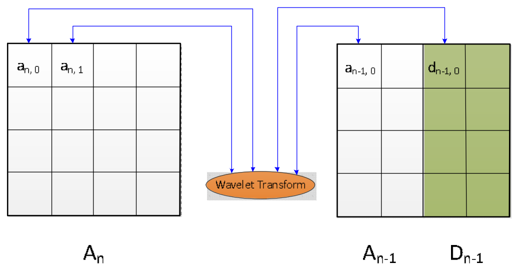


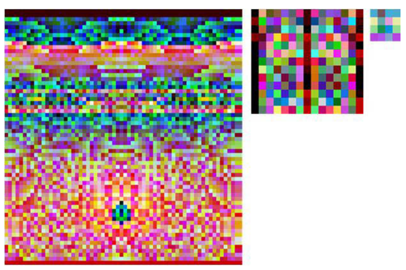
Fig. 11: GPU wavelet transform from A_n to A_{n-1}.

The pseudo code below describes the wavelet transform process, the *AnTexArray* represents the NURBS texture array data A_n. With texture2Drect, $a_{n,0}$ and $a_{n,1}$ can be fetched from the NURBS texture array data. Then wavelet transform is applied, and the results are written to gl_FragData[0] and gl_FragData[1], which represent the the main part of A_{n-1} and the detailed part of D_{n-1}. Similarly, the reverse transform generates A_n from A_{n-1} and D_{n-1}.

```
é
uniform sampler2DRect AnTexArray;
vec2 coords = gl_TexCoord[0].xy;

vec4 a_n0 = texture2DRect(AnTexArray, vec2(coords.x*2.0-0.5,coords.y));
vec4 a_n1 = texture2DRect(AnTexArray, vec2(coords.x*2.0+0.5,coords.y));
gl_FragData[0] = 0.5*( a_n0 +a_n1);
gl_FragData[1] = 0.5*( a_n0 ða_n1);
é
```

The wavelet transform is simple and consume little time compared with NURBS re-evaluation method that needs to re-evaluate the NURBS curves and surfaces, because it can generate A_{n-1} form A_n and retrieve A_n from A_{n-1} and D_{n-1} by reverse. So this multi-resolution LOD method is somewhat like the progressive meshes proposed by Hoppe [10]. Different texture arrays corresponding to the LOD levels are the result of wavelet transform, namely A_n and A_{n-1} shown in Fig. 12 (a). Compared with progressive meshes, the wavelet transform progress can also be accelerated by GPU shader in parallel. The wavelet multi-resolution analysis based LOD rendering effects is shown in Fig. 12 (b).



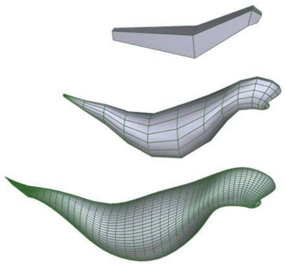


Fig. 12: Wavelet multi-resolution analysis based LOD: (a) wavelet transform result of *S(u,v)* texture array data, and (b) rendering effects of wavelet based LOD.

5 RESULTS AND ANALYSIS

We carried out test of GPU de Boor's NURBS evaluation method and GPU LOD rendering method on different hardware platforms with Intel CPU and NVIDIA GPU shown in Tab. 1.

CPU/GHz	*RAM/GB*	*GPU*	*VRAM/MB*
P4 3.0	2	GeForce 9500GT	512
Core2 2.13*2	2	GeForce GT 130M	512
Xeon 2.67*8	16	Quadro FX 3800	1024

Tab. 1: Hardware platforms and configuration.

NURBS curve and surface evaluation speed is affected by order, number of evaluation points and control points. In practice, the order is usually no more than 3, so we do not need to test our method with order, we directly choose order 3. The evaluated NURBS surfaces are shown in Fig. 13.

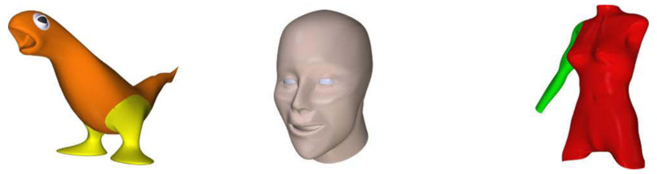


Fig. 13: NURBS surfaces evaluated and rendered on GPU.

Fig. 14 is the comparison of GPU De Boor's evaluation method and basis function method of evaluation time compared with [7]. We test evaluation with control points of 16*16, 64*64 and 256*256. The figure shows that our method promotes the evaluation speed orders of magnitude. The main difference is because our De Boor's algorithm carries out everything by GPU parallel computing with one pass while GPU basis function method needs to compute different order of basis function with multi pass. Meanwhile, our De Boor's algorithm uses NURBS local characteristic and only uses control points and knots vector that effects the evaluation point described in 3.2, while their method compute the whole control points with knots vector.

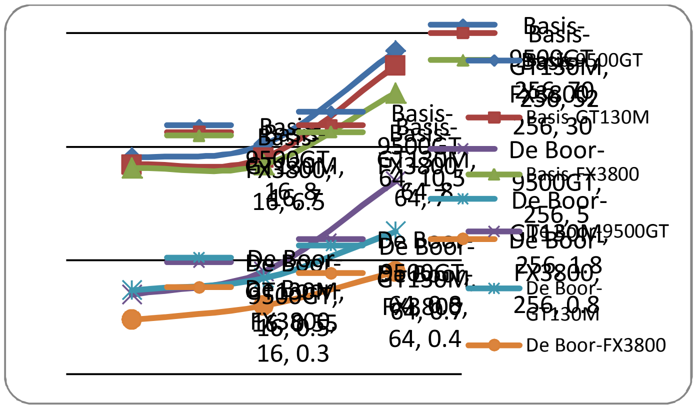


Fig. 14: Comparison of evaluation speed of De Boor's method with basis function method on different GPU.

Fig. 15 is the comparison of LOD rendering method of GPU wavelet transform based multi-resolution analyze with traditional rendering method. We test evaluation points of 128*128, 512*512 and 1024*1024 with control points of 21*23. The figure shows that rendering with wavelet transform is 3-5 times faster than re-evaluation because it generates A_n from A_{n-1} and D_{n-1} by gradually refine, not totally discard previous evaluated data. So it has little overload to both CPU and GPU when changing the LOD detail and NURBS rending resolution.

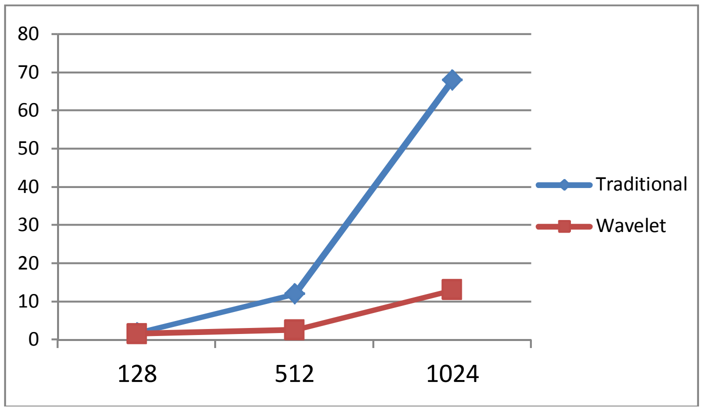


Fig. 15: Comparison of LOD rendering by wavelet transform of multi-resolution analysis and traditional LOD rendering.

6 SUMMARY AND CONCLUSIONS

We have implemented NURBS De Boor's algorithm on the GPU, which promote NURBS evaluation speed orders of magnitude compared with previous GPU basis function method. With vertex texture fetch technology of recent GPU, we demonstrate the wavelet transform of multi-resolution based LOD rendering with the evaluated texture remain in GPU and the connectivity index data complied to display list, so GPU accelerated rendering have almost no overload to CPU and GPU.

ACKNOWLEDGEMENTS

This research is supported by Program for New Century Excellent Talents in University (The project NO. is NCET-10-0036), State Key Laboratory of Virtual Reality Technology and Systems (The project NO. is SKVR-09-08) and Fund of National Engineering and Research Center for Commercial Aircraft Manufacturing，(The project No. is SAMC-11-JS-07-203）.

REFERENCES

[1] Piegl, L.A.; Tiller, W.: The NURBS book, Springer Verlag, 1997.

[2] Dietrich, A.; Wald, I.; Slusallek, P: Large-scale CAD model visualization on a scalable shared-memory architecture, Proceedings of 10th International Fall Workshop-Vision, 2005, 303ð310.

[3] Patidar, S.; Bhattacharjee, S.; Singh, J.M.; Narayanan, PJ: Exploiting the shader model 4.0 architecture, Center for Visual Information Technology, 2007.

[4] Kanai, T.; Yasui, Y.: Per-pixel evaluation of parametric surfaces on gpu, ACM Workshop on General Purpose Computing Using Graphics Processors, 2004.

[5] Guthe, M.; Balazs, A.; Klein, R.: GPU-based trimming and tessellation of NURBS and T-Spline surfaces, ACM Transactions on Graphics (TOG), 24(3), 2005, 1016ð1023. DOI:10.1145/1073204.1073305

[6] Kanai, T.: Fragment-based evaluation of Non-Uniform B-spline surfaces on GPUs, Computer-Aided Design and Applications, 4(3) , 2007, 287ð294.

[7] Krishnamurthy, A.; Khardekar, R.; McMains, S.: Direct evaluation of NURBS curves and surfaces on the GPU, Proceedings of the 2007 ACM symposium on Solid and physical modeling, 2007, 329ð334. DOI:10.1145/1236246.1236293

[8] Krishnamurthy, A.; Khardekar, R.; McMains, S.: Optimized GPU evaluation of arbitrary degree NURBS curves and surfaces, Computer-Aided Design, 41(12), 2009, 971ð980. DOI:10.1016/j.cad.2009.06.015

[9] Luebke, D.P.: Level of detail for 3D graphics, Morgan Kaufmann Pub, 2003.

[10] Hoppe, H.: Progressive meshes, Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, 1996, 99ð108.

[11] Rockwood, A.; Heaton, K.; Davis, T.: Real-time rendering of trimmed surfaces, Proceedings of the 16th annual conference on Computer graphics and interactive techniques, 1989, 107ð116.

[12] Han, D.: Tessellating and Rendering Bezier/B-Spline/NURBS Curves and Surfaces using Geometry Shader in GPU. Citeseer, 2008, http://www.gpucomputing.net/?q=node/2934.

[13] Carl De Boor, A.: A practical guide to splines, Springer-Verlag, 1978.

[14] Goddeke, D.: Gpgpu-basic math tutorial, Univ. Dortmund, Fachbereich Mathematik, 2005, http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html.

[15] Lindstrom, P.; Koller, D.; Ribarsky, W.; Hodges, L.F.; Faust, N.; Turner, G.A.: Real-time, continuous level of detail rendering of height fields, Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, 1996, 109ð118.