Exploiting the Shader Model 4.0 Architecture

Suryakant Patidar* Shiben Bhattacharjee† Jag Mohan Singh‡ P. J. Narayanan§
Center for Visual Information Technology, IIIT Hyderabad

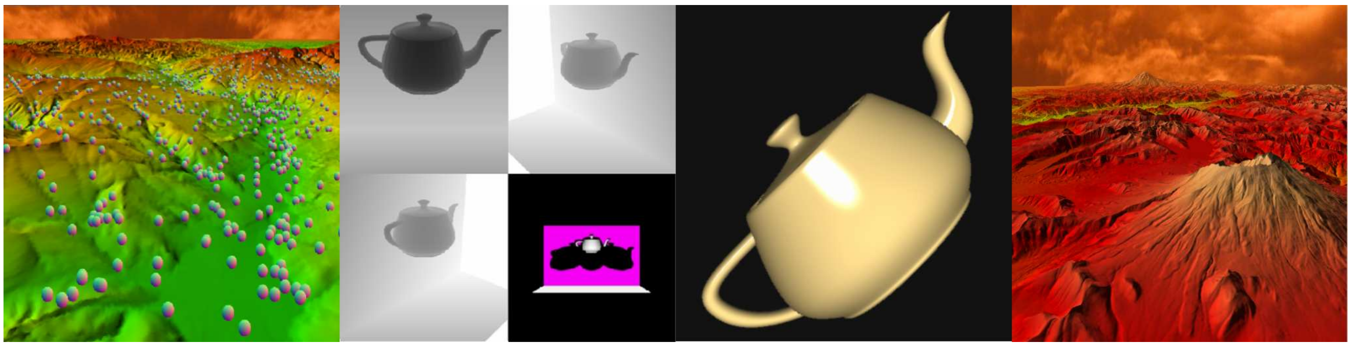

Figure 1: *Physics of balls with Transform Feedback. Multi Light Shadows with Layered Rendering. Teapot rendered with spline subdivision. Geometry generation with Geometry Shader.*

Abstract

The Direct3D10/SM4.0 system [Blythe 2006] is the 4^{th} generation programmable graphics processing units (GPUs) architecture. The new pipeline introduces significant additions and changes to prior generation pipeline. We explore these new features and experiment to judge their performance. The main facilities introduced that we ponder upon are, Unified Architecture providing common features set for all programmable stages, Geometry Shader which is a new programmable stage capable of generating additional primitives, Stream output with which primitive data can be streamed to memory, Array textures and primitive level redirection to different frame buffers through layered rendering. We analyze our implementations and with experimentation, we draw conclusions on their efficient usage and provide some of their limitations.

1 Introduction

In real-time 3D computer graphics, a *rendering pipeline* is meant to accept certain representation of a 3D scene as input and produce a 2D raster image as output which can be shown on a display device. The various *pipeline stages* are mentioned in Figure 2. Earlier, these stages were implemented as fixed functions on hardware for graphics acceleration. Later, some of these stages were made programmable instead of being fixed function namely processing of vertices and fragment. Since all vertices and fragments can be thought of as independent, they can be processed parallelly on the graphics processing unit. Small programs could be written which processed all vertices (or fragments) parallelly and were called *shaders* and the programming model was called the *shader model*. The history of graphics hardware and shader model is discussed in Section 1.1.

*e-mail:skp@research.iiit.ac.in
†e-mail:shiben@research.iiit.ac.in
‡email:jagmohan@research.iiit.ac.in
§e-mail:pjn@iiit.ac.in

The Shader Model 4.0 makes giant leaps considering a version change, as it involves significant changes to the rendering urepipeline. Earlier vertex shaders and fragment shader, being at different stages, had different requirements. Fragments shaders had quick access to textures and vertex shaders either didn't have or had limited access to textures. Shader Model 4.0 keeps all the shaders at the same level calling it a unified architecture, because of which even vertex shader (and geometry shader) have fast access to textures. This feature can be used to draw objects whose geometry information is stored as images, a very good example being terrain rendering in which elevation map is stored as images. We discuss usage of unified architecture in Section 2.

The new model introduces a new programmable stage called the *geometry shader* which gives the capability of generating more vertices and primitives. This saves the CPU to GPU bandwidth as less 3D data can be sent using programming tricks to the pipeline and they can be generated on the GPU itself. We stress test the capabilities and verify the importance of this new programmable stage (Section 3). We develop techniques and applications involving geometry shader and our analysis will be helpful in clever usage of geometry shader.

In past, game developers complained about slow state change because of which they avoided using multiple textures. Instead they mapped textures on various objects with the help of a single texture, texture atlas [Nvidia 2004]. Shader Model 4.0 introduces array textures and 3D textures. Various images can be loaded as layers in array texture (or as a 3D texture) and the shader can choose between these layers as the whole array texture is referenced by a single texture ID. Terrain rendering finds itself as a good application of array textures which we explain in detail in Section 4.

Along with the introduction of array textures, another new attractive feature is layered rendering which seems an extension of older model's one of the feature, *MRT (Multiple Rendering Targets)*. Where with MRT, fragments in the fragment shader can be redirected to different *color attachments* of the frame buffer; with *layered rendering*, it is possible in the geometry shader to redirect primitives to different layers of an array texture which are attached to the frame buffer object. Independent rasterization for all the dif-

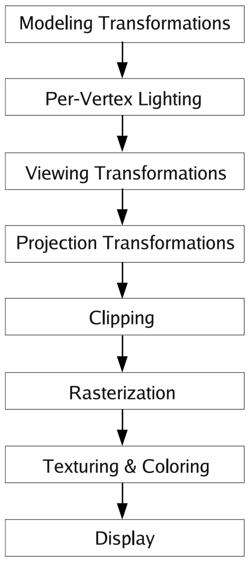

Figure 2: *Traditional rendering pipeline*

ferent layers will be carried out. We find that multi-perspective rendering and multiple light shadows are good applications of layered rendering and are discussed in Section 5. Real-time Cube-map rendering is good application which has been discussed in [Blythe 2006].

The new rendering pipeline also features streaming data out in the middle of the pipeline to the memory. This is termed as *stream output* by the Direct3D 10 system and *transform feedback* by the OpenGL 2.1 system. This gives the facility of not going through the whole pipeline to process stream data which is not to be rendered. Objects can be rendered as well as some information can be recorded in the same pipeline. We implemented simple physics transformations and deformable models which harnesses the transform feedback's facilities; they are described in Section 6.

A small overview of the features which came along with shader model 4.0 as enumerated below:

1. Unified Architecture : A single point computation unit which is responsible for the variety of computations going on the GPU (Section 2).
2. Integer Operations : Integer (arithmetic, bitwise and conversion) instructions (32-bit) are now supported.
3. Goemetry Shader : A new shader unit in the pipeline which takes a primitive (point, line or triangle) as input and produces *zero* or more primitives (Section 3).
4. Array Textures : This feature of Shader Model 4.0 brings along two new texture storage units namely *one- and two-dimensional array textures* which essentially mean a collection of one- and two-dimensional textures (Section 4).
5. Layered Rendering : With the introduction of Geometry Shader user can render to one of several different layers of cube map textures, three-dimensional textures, plus one- and two-dimensional texture arrays (Section 5).
6. Stream Output : This new mode to the graphics pipeline records vertex attributed of the primitives processed in the geometry shader or post vertex shader in case of fixed pipeline or absence of geometry shader (Section 6).

1.1 Programmable GPUs : A Brief History

In history, 3D graphics goes as early as the days of Evans & Sutherland, founded by Ivan Sutherland and David Evans in 1968. Some of their work involves projects for military and large industrial firms for training and simulation, digital projection environments like planetariums etc. SGI was founded by one of the students of Ivan Sutherland, Jim Clark and Abbey Silverstone in 1981. In 1982, one of the first products from SGI include IRIS 1000 (Integrated Raster Imaging System). SGI also came up with an API, 'IRIS Graphics Language' (IRIS GL), which provided access to their high performance 3D graphics subsystems. Later, IRIS GL became OpenGL which meant for the first time, fast, efficient, cross-platform graphics programs could be written.

Commodity based graphics hardware involving 3D functions were first developed by Matrox, Creative, S3 and ATI in 1995. Graphics hardware production for consumer PC market grew with the introduction of Voodoo Graphics PCI by 3Dfx in 1996. 3dfx also provided a graphics API called GLIDE for their VooDoo cards, which was kept at a much lower level to the hardware. This was difficult for game programmers but they were able to harness the full power of voodoo cards. The first voodoo card was capable of only rasterization and pixel processing. Later NVIDIA developed the first card named *GeForce 256 (SDR)* in Oct 1999. This card was capable of fixed function hardware based transformation of vertices and lighting with the Direct3D7 and OpenGL1.2 API.

The GeForce 3, was the first programmable GPU, allowing game developers to do much more (2001). This breakthrough came as Direct3D8 introduced the first widely-used family of shaders: Shader Model 1.1. ATI came up with Radeon 8500 supporting Shader Model 1.4. The shader model at this stage involved a few of assembly intructions in the shader programs. Also branching and looping were not supported at all. The next version, Shader Model 2.0 (Direct3D9), involved improvement in terms of shader code length. GeForceFX series and Radeon 9700 brought it to the consumers in 2003.

In 2005, we had the GeForce 6 and 7 series, which support Direct3D9c and Shader Model 3.0. X800 supported Direct3D9c but only Shader Model 2.0b. Shader Model 3.0 also saw the dawn of high level shading languages. Direct3D9c named its shading language HLSL and OpenGL2.0 called it GLSL1.1. Shader Model 3.0 contained several improvements: improved floating point precision, more code length, support of dynamic branching but with a performance hit, unrolled looping and vertex texture lookup. Due to such enhancements, Shader Model 3.0 remained the most used by game developers, graphics researchers and GPGPU programmers. NVIDIA released its own shading language Cg in 2003, which was closer to HLSL, and was apparently used widely.

By the end of year 2006, NVIDIA released the first card, 8800GTX, of their G80 family. This supported Shader Model 4.0 introduced by Direct3D10 System. OpenGL 2.1 with GLSL1.20 got introduced in November 2006 which fully supported the new features of Shader Model 4.0. AMD is expected to release its R600 series card, Radeon 2900XT, in May 2007. The improvements in Shader Model 4.0 are significant and tend to solve most issues game developers raise.

$CUDA^{TM}$ (Compute Unified Device Architecture) which is a fundamentally new computing architecture, provides an access to the tremendous processing power of NVIDIA GPUs through a revolutionary new programming interface. The core technology of GPUs which is *parallel data processing* is now exploited with GPUs entering the space of massively parallel processing. The GPUs now act as a co-processor to the CPU, offloading major part of process-

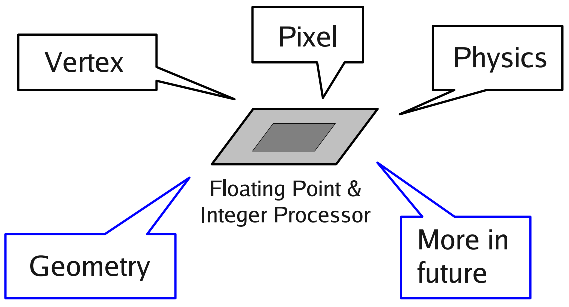

Figure 3: *The Unified Architecture*

ing.

2 Unified Shader Architecture

The Shader Model 4.0 GPUs use the same processor core to implement vertex, geometry and fragment processing. Seperate processors with different capabilities were used for different stages of the earlier GPUs. The previous model with seperate processors for vertex and pixel units (Figure 4) was prone to under-performance. The Shader Model 4.0 GPUs dynamically allocate the available processing resources to vertex, geometry and pixel units as demanded by the load. This greatly improves the resource utilization.

The main advantages of Unified Architecture are

1. Dynamic load balancing due to the on-demand scheduling of processes
2. Higher power to vertex processor with introduction of full floating point precision
3. High performance due to unified compute and texture access.

An important advantage of unified architecture is uniform access to the texture memory for all shaders. The vertex textures supported on Shader Model 3.0 were slow [Asirvatham and Hoppe 2005]. ATI cards never supported vertex textures. This allows the use of textures to store regular geometry and their access in vertex and geometry shaders. We compare rendering a grid of 1000 $\times$ 1000 points using vertex texture to fetch the vertices. The FPS on a Nvidia 6600 GT was 14, and on a Nvidia 7950GX2 was 32 fps. In contrast GeForce 8800 GTX gave an 270 fps. This together with geometry generation on the Geommetry Shader (Section 3.1) makes it possible to render huge geometry completely on the shaders. Some of the applications which can take direct advantage of the above scenario are rendering of Geometry Images and rendering of huge terrains with dynamic LOD. Thus, unified shaders allow us to store geometry in form of textures and efficiently access the same from vertex shader.

These features make it possible for the whole geometry to be stored in and rendered from the the GPU memory. This can increase the rendering speed especially if the geometry is quite regular. Terrains are good examples of such geometry when represented using regular heightmaps. The 2D array of heights can be stored in textures on the GPU and accessed by the shaders quickly for rendering.

Gu et al introduced Geometry Images [Gu et al. 2002] which captures geometry as a simple 2D array of quantized points. As opposed to remeshing an irregular mesh into one with a semi-regular connectivity, they proposed a technique to remesh an arbitrary surface onto a completely regular structure called a geometry image. With the introduction of faster (common to vertex, geometry and

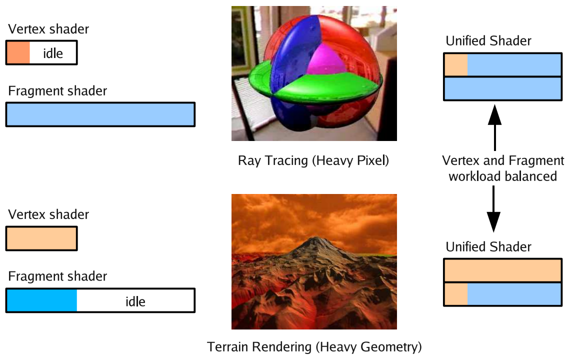
Vertex shader
idle
Fragment shader
Ray Tracing (Heavy Pixel)
Unified Shader
Vertex and Fragment workload balanced
Vertex shader
Fragment shader
idle
Terrain Rendering (Heavy Geometry)
Unified Shader

Figure 4: *Inefficient usage of hardware with previous architecture. Schematic representation of dynamic load balancing with Unified Architecture*

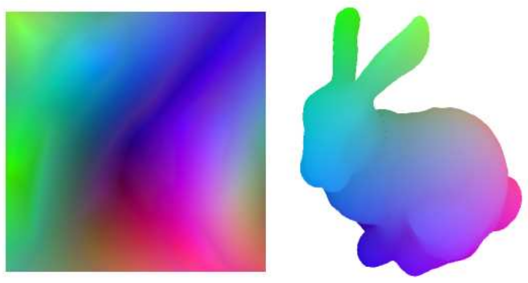

Figure 5: *Rendering Geometry Images with vertex textures*

fragment shader) texture fetch in the shaders a geometry image can be rendered with high efficiency. For a geometry image of Bunny with 257 $\times$ 257 vertices the fps obtained on a Nvidia 7950GX2 was 713 as compared to that on 8800 GTX was 2281 (Figure 5).

Code Snippet from GLSL geometry shader for Geometry Image rendering.

```
...

// pos *= 5.0 corresponds to the scaling
// factor as normalized vertices are stored in
// the Geometry Image.

// tex is the texture holding the geometry
// image.

pos.xyz = texture2DRect( tex, texcoord.xy ).rgb;
pos.xyz *= 5.0;
pos.w = 1.0;
gl_Position = gl_ModelViewProjectionMatrix * pos;
EmitVertex();

texcoord.x += 1.0;
pos.xyz = texture2DRect( tex, texcoord.xy ).rgb;
pos.xyz *= 5.0;
pos.w = 1.0;
gl_Position = gl_ModelViewProjectionMatrix * pos;
EmitVertex();

texcoord.x -= 1.0;
texcoord.y += 1.0;
pos.xyz = texture2DRect( tex, texcoord.xy ).rgb;
pos.xyz *= 5.0;
pos.w = 1.0;
gl_Position = gl_ModelViewProjectionMatrix * pos;
EmitVertex();

texcoord.x += 1.0;
pos.xyz = texture2DRect( tex, texcoord.xy ).rgb;
pos.xyz *= 5.0;
pos.w = 1.0;
gl_Position = gl_ModelViewProjectionMatrix * pos;
EmitVertex();

EndPrimitive();
...
```

The above code corresponds to expansion of a point from geometry image into a 2×2 grid of triangle to build the mesh.

Following 3 tables show the FPS numbers under various conditions.

[1] Geometry being sent from CPU in form of Points
[2] Geometry being sent from CPU in form of Triangles (TRIANGLE_STRIP)
[3] Dummy Indices stored in VBO, expanded in shader and heights fetched from vertex texture(LUMINANCE)
[4] Dummy Indices stored in VBO, expanded as triangles in Geometry Shader and heights fetched from texture

Nvidia 6600 GT			
Grid Size	CPU(Pts) [1]	CPU(Tri) [2]	VBO+VT(Pts) [3]
256 × 256	259	155	200
512 × 512	65	39	80
1024 × 1024	17	10	23
2048 × 2048	5	3	6

Nvidia 7950 GX2			
Grid Size	CPU(Pts) [1]	CPU(Tri) [2]	VBO+VT(Pts) [3]
256 × 256	370	180	740
512 × 512	94	50	215
1024 × 1024	24	13	59
2048 × 2048	6	3	15

Nvidia 8800 GTX				
Grid Size	CPU(Pt)[1]	CPU(Tr)[2]	VT(Pt)[3]	VT(Tr)[4]
256 × 256	705	374	1851	731
512 × 512	177	98	905	284
1024 × 1024	44	24	261	90
2048 × 2048	13	7	68	27

Following table contains the FPS numbers for rendering of Geometry Images. We have used the freely available Geometry Image of the Bunny available at Hoppe's web site.

[1] Rendering Points
[2] Rendering Triangles with help of Geometry generation

6600 GT [1]	7950 GX2 [1]	8800 GTX [1]	8800 GTX [2]
271	713	2281	976

3 Geometry Shader

Geometry Shader is the most notable new feature of Shader Model 4.0. This defines a new shader type which runs on the Graphics Processing Unit. Geometry shaders are invoked after vertices are processed by vertex shader, but prior to color clamping, flat shading and clipping (Figure 6). The input to the Geometry Shader is a single primitive (point, line, triangle) and other attributes like texture coordinates, color, normal, etc. The output is a list of primitives, whose number, type, and attributes need not match with those of the input primitives. The Geometry Shader can use all input information and data stored in textures to generate its output. It can, therefore, discard the geometry in the pipeline or insert new geometry into it. The new and potentially disconnected primitives generated by the geometry shader are treated like other primitives coming directly from the application (OpenGL/DirectX).

Input/Output to the geometry shader and maximum number of vertices emitted per call of geometry shader can be mentioned in the following way from an OpenGL code.

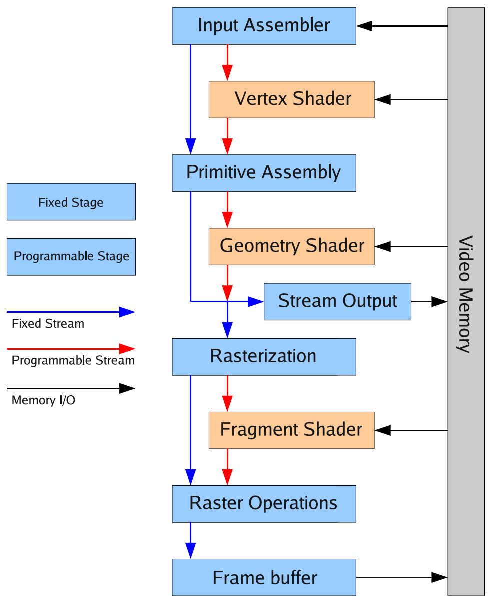

Figure 6: *The Shader Model 4.0 Graphics Pipeline*

```
...
//Informing the Geometry Shader that there will be 4
//vertices emitted per primitive

glProgramParameteriEXT ( shader,
GL_GEOMETRY_VERTICES_OUT_EXT, 4 );

//Informing the Geometry Shader that the input type to
//GS will be Points

glProgramParameteriEXT ( shader,
GL_GEOMETRY_INPUT_TYPE_EXT, GL_POINTS );

//Informing the Geometry Shader that the Output type to
//GS will be Triangle Strips

glProgramParameteriEXT ( shader,
GL_GEOMETRY_OUTPUT_TYPE_EXT, GL_TRIANGLE_STRIP );
...
```

Geometry shader can also access the neighbouring vertices of a primitive. The above scenario called for introduction of a category of primitives viz. lines with adjacency (LINES_ADJACENCY_EXT), line strips with adjacency (LINE_STRIP_ADJACENCY_EXT), triangles with adjacency (TRIANGLES_ADJACENCY_EXT), and triangle strips with adjacency (TRIANGLE_STRIP_ADJACENCY_EXT). These primitives take in more than the usual number of vertices; a line with adjacency requires 4 vertices to describe its end points and their neighbours and a triangle with adjacency needs 6. Thus, at the cost of extra geometry transfer, one can access the important information of the neighbourhood of a primitive in the geometry shader. Silhouette calculation of 3D models requires neighbouring information for the calculation of gradients. Thus, one can use TRIANGLES_ADJACENCY_EXT primitives in order to provide the neighbourhood vertices to be processed in the geometry shader. Geometry shader can then access a couple of extra vertices and find the required information (normals and tangents).

3.1 Geometry Generation

We describe an example of genertion and expanding of geometry by the Geometry Shader in this section by rendering terrains of various size with diffrent amount of geometry being sent and rest of it being generated on the Geometry Shader. With the driver version 1.0-9755 for Linux the maximum amount of geometry which can be generated is capped at 128 vertices per Geometry Shader execution..

Grid Size	Geom. Sent[1]	Geom. Gen.[2]	FPS
512×512	512×512	2×2	203
512×512	256×256	4×4	91
512×512	128×128	8×8	21
1024×1024	1024×1024	2×2	51
1024×1024	512×512	4×4	23
1024×1024	256×256	8×8	6

Grid Size	Geom. in VBO[3]	Geom. Gen.[2]	FPS
512×512	512×512	2×2	213
512×512	256×256	4×4	91
512×512	128×128	8×8	21
1024×1024	1024×1024	2×2	60
1024×1024	512×512	4×4	23
1024×1024	256×256	8×8	6

1 Amount of Geometry being sent from CPU (as Points)

2 Amount of Geometry generated in Geometry Shader to form the Grid

3 Amount of Geometry stored in VBO

Shader Model 4 Terrain System is a completely GPU based terrain system where most of processing is done on the geometry shader with terrain geometry being accesed from the textures (2D Array Textures), tiles (grid of triangles) generated and expanded with some information from small VBOs and textures containing geometry and stitching of these tiles. The amount of geometry which can be generated on Geometry Shader is limited. Currently on a Nvidia 8800GTX we are able to expand a vertex to 128 vertices. With this constraint we render a 257×257 tile (with an extra row and coloumn for stitching at top and right borders) with various VBO sizes of 32×32, 64×64 etc. Further these points from the VBOs are expanded as grids to fill in for the missing near by points.

With the introduction of Geometry Shader one can send in dummy points which can be further expanded as a grid of triangles in the geometry shader. A novel approach for two level culling (Section 3.2) was used in the Terrain System which performs a second level culling algorithm on the incoming points to the vertex shader and culls the vertices which would lie outside the frustum after going through a Modelview and Projection. Thus for each tile we would perform a tile-level culling on the CPU and set a flag for the tiles which are intersecting the frustum. Later in the vertex shader we receive a $m \times n$ grid of points for a tile of size $M \times N (m < M; n < N)$, for each of these incoming points we pass them through the second level culling test, thus avoiding a lot of expansion and rendering of extra geometry.

3.2 Two Level Culling

Geometry deletion enables culling of geometry in the geometry shader. The tiles in the terrain system undergo the first level of culling on the CPU where the CPU culls large tiles and marks the ones which intersect the frustum. The marked tiles are then tested for further culling on the GPU in vertex shader. For each tile of size 256×256 to be rendered (Figure 7), CPU sends a VBO of size $m \times n$ where $m < 256, n < 256$ (in case of Nvidia 8800GTX, $32 < m < 256, 32 < n < 256$, where maximum geometry generation (8×8) happens with $m = 32, n = 32$). Each of these points in the grid of $m \times n$ are then tested in the vertex shader if they lie inside or outside the frustum and accordingly labelled as culled or not culled. We save the geometry generation and rendering load for these points by culling them in vertex shader. The culling on the

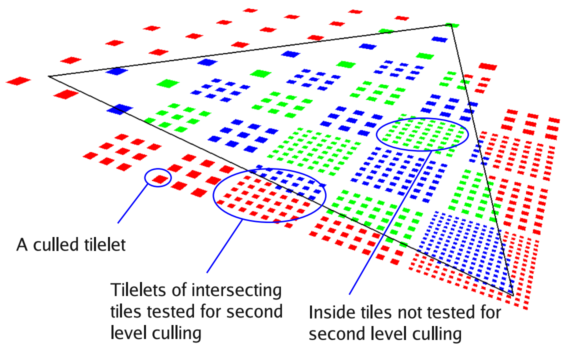
A culled tilelet
Tilelets of intersecting tiles tested for second level culling
Inside tiles not tested for second level culling

Figure 7: *Tiles and Tilelets involved in first and second level of culling*

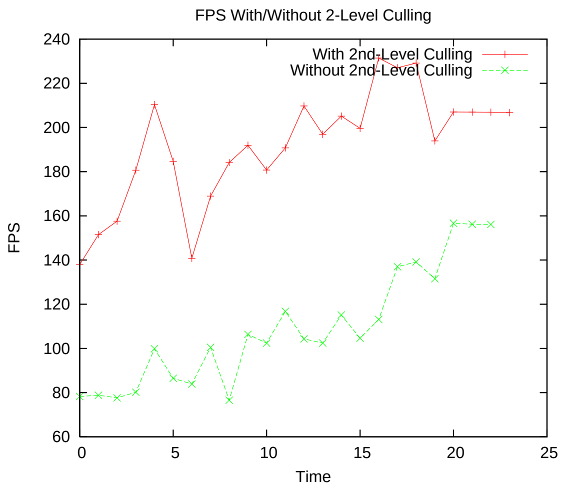

Figure 8: *Two level culling performance against generating and rendering the extra geometry left out by the CPU culling*

vertex shader is performed by mapping these grid points to screen space and testing them against a range. The main advantage of the technique is that we render exact amount of geometry required, thus an exact culling is performed. Secondly we can reduce the CPU load by increasing the size of tiles.

3.3 Subdivision on the Geometry Shader

The geometry shader can be used to increase/decrease the number of primitives which are passed on by the vertex shader to the geometry shader. An application in which the number of vertices increase with the iterations could be easily ported on GPU. Subdivision of spline patches is one such application. The subdivision is done in *s-t* space for the input triangles. Our current scheme of subdivision divides a square into a grid of size $2^k \times 2^k$ for subdivision of level k. Since, our input is only a triangle we take the lower half of the grid. Thus, for doing subdivision on GPU given an input triangle, we generate the set of vertices on the grid and output the appropriate set of triangles to the pixel shader. The pixel shader then evaluates the position and normal at the interpolated s-t coordinate and does per-pixel lighting and depth computation.

We calculate the position and normal at a given s-t using vector dot products as shown in the following code snippet.

```
// Calculation of position and normal at s-t when
```

```
geometry vectors are gx,gy and gz for the bicubic patch
float s2 = s*s;
float s3 = s2*s;
float t2 = t*t;
float t3 = t2*t;
vec4 svec = vec4(s3,s2,s,1.0);
vec4 tvec = vec4(t3,t2,t,1.0);
vec4 ssvec = vec4(3.0*s2,2.0*s,1.0,0.0);
vec4 ttvec = vec4(3.0*t2,2.0*t,1.0,0.0);
stangent.x = dot(ssvec*gx,tvec);
..
ttangent.x = dot(svec*gx,ttvec);
...
normal = cross(stangent,ttangent);
position.x = dot(svec*gx,tvec);
...
// Vertices generated for 2 levels of subdivision
// Input triangle with it's s-t coordinates
vec4 a = gl_PositionIn[0];
vec4 b = gl_PositionIn[1];
vec4 c = gl_PositionIn[2];
//The grid spacing for the subdivided triangle
vec4 dx = (b-a)/4.0;
vec4 dy = (c-a)/4.0;
//s-t coordinates of the output triangles
for(int i=0;i<5;i++)
for(int j=0;j<5;j++)
vertices[5*i+j] = a+dx*j+dy*i;
// Outputting a triangle
// s-t coordinate of the triangle
gl_TexCoord[0] = vertices[i];
//Calculate position,normal and color at vertex
//using these s-t coordinates
gl_FrontColor = outcolor;
//Output fragment
gl_Position = gl_ModelViewProjectionMatrix*outposition;
//Signal end of a vertex
EmitVertex();
//Similarly for other two vertices
.... ...
// Signal end of the triangle
EndPrimitive();
//Similarly output other triangles
... ....
```

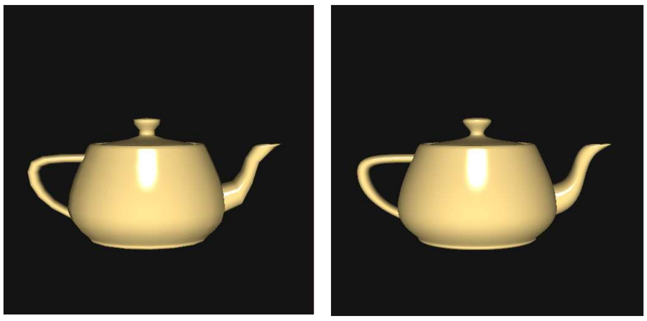
Figure 9: *Spline teapot with no subdivision on left and 2 levels of subdivisions on right*

Currently there are limitations on the number of vertices that can be generated by the geometry shader which limits us to 2 levels of subdivisions, i.e., each input triangle can be used to generate 16 triangles. We show our results on teapot which consists of 32 bicubic spline patches. Figure 9 shows the teapot with no subdivisions on the left and two levels of subdivision on right. The triangulation artifacts are less apparent in the right one. The FPS for the teapot with no subdivisions but per-pixel lighting and depth computation is 750 and that for two levels of subdivision is 140 at 512×512 resolution on NVIDIA 8800.

4 Array Textures

One of the extensions introduced with Shader Model 4.0 is EXT_texture_array. This extensions introduces the idea of one- and two-dimensional array textures. A Texture array is a collection of one- and two-dimensional images of identical size and format, organized in layers. *TexImage2D* is used to specify an one-dimensional array texture, where the *height* specify the number of layers for the array texture. Similarly *TexImage3D* is used to declare a two-dimensional array texture, where the *depth* specify the number of layers of 2D textures. Maximum number of layers which can be attached together can be queried using GetIntegerv with pname as MAX_ARRAY_TEXTURE_LAYERS_EXT. On a nVIDIA 8800 GTX the maximum number of layers possible is 512.

2D-Array Textures are declared as follows :

```
//Generate an opengl texture
glGenTextures(1,&texid);
glBindTexture(GL_TEXTURE_2D_ARRAY_EXT,texid);

//Setting up the parameters, note the 'R' space as well
glTexParameteri( GL_TEXTURE_2D_ARRAY_EXT,
  GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D_ARRAY_EXT,
  GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D_ARRAY_EXT,
  GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
glTexParameteri( GL_TEXTURE_2D_ARRAY_EXT,
  GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
glTexParameteri( GL_TEXTURE_2D_ARRAY_EXT,
  GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE );

//Create space for the array texture
glTexImage3D( GL_TEXTURE_2D_ARRAY_EXT, 0, GL_RGBA, width,
  height, depth, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);

//Put individual 2D image data to each layer
glTexSubImage3D( GL_TEXTURE_2D_ARRAY_EXT, 0, 0, 0, layer,
  width, height, 1, GL_RGBA, GL_UNSIGNED_BYTE,
  image_data_pointer);
```

Texture array can be used in a shader as :

```
//declaration of sampler
uniform sampler2DArray tex;
...
//fetching color from layer texcoord.z from the 2D
//location texcoord.xy from sampler terraintex
color = texture2DArray ( terraintex, texcoord ).rgba;
...
```

Currently, this extension is not supported on the fixed-function fragment processing. Array textures can be accessed only using the programmable shaders. A layer is selected by specifying the *t* or *r* texture coordinate for 1D and 2D array textures, respectively. Further it is accessed as though it were a one- or two-dimensional texture. A 2D array texture sounds similar to a 3D texture. There are few differences between 2D array textures and 3D textures. In case of 2D array textures, texture lookups do not filter between layers, though the same can be achieved in the shaders, where as in case of 3D textures various filtering options are provided. Rendering to different layers or 2D textures can only be achieved by binding a 2D array texture to a frame buffer object.

Texture arrays are ideal to store data which is independent at each layers, for e.g. terrain data. Terrain can be divided into blocks of desired size $(1024 \times 1024, 2048 \times 2048)$ and these blocks can be stored as various layers in a 2D array texture. An array texture is accessed as a single unit in a programmable shader, using a single coordinate vector. A terrain extending over $m \times n$ blocks can be stored as an array texture with mn layers, giving the user a single point access to the whole or currently cached terrain. Several computations like physics calculation (Section 6.1) need to have access to whole of the terrain. With array textures any part of the

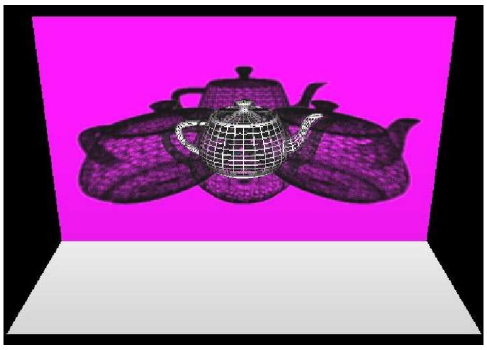

Figure 10: *Multiple Shadows in 2 Pass.*

terrain can be accessed by specifying the layer with *r* texture coordinate in case of a 2D array textures, thus providing the ease of selecting textures in the programmable shaders rather than an intervention from CPU regarding the binding of correct texture. The same was possible with 3D textures, but the array textures can additionaly be rendered to by binding them to a frame buffer object (EXT_framebuffer_object). Along with physics, a dynamic terrain could itself undergo deformation on the fly (may be an impact from the ball can produce craters). A continously changing terrain which is stored as an array texture(s) can be evolved by rendering to these layers [Bhattacharjee et al. 2007].

5 Layered Rendering

The main use of 2D array texture comes with Layered Rendering. With an array texture bound to a framebuffer object we can route geometry from geometry shader to different layers. Thus rendering different scenes/views in a single pass becomes possible. Multiple Render Targets were introduced in order to render to more than one framebuffers at Fragment Shader level. With MRT, we can produce more than one output at the pixel level. We can say Layered Rendering provides the freedom of doing the same thing at geometry level and each layer can have independent rasterization.

In case of dynamic Environment Mapping, i.e., when the Environment Map is not static and changes with camera motion, we can produce all the six faces of the map in a single pass thus giving a 2-pass algorithm for Cube-Map rendering. A couple of more applications of Layered Rendering which are also described below are, two pass motion blur and 2 pass multiple lights dynamic shadows. To bind the layers of an array texture to a frame buffer object, following code can be used.

```
...
//Generate Texture to hold the 2D Array Texture
glGenTextures( 1, &colorTex );
glBindTexture( GL_TEXTURE_2D_ARRAY_EXT, colorTex );

//Create a 2D array texture and set other properties
glTexImage3D ( GL_TEXTURE_2D_ARRAY_EXT, 0, GL_RGBA8, 256,
  256, 2, 0, GL_RGBA, GL_FLOAT, 0 );
...
//Generate an FBO
glGenFramebuffersEXT( 1, &fbo );
glBindFramebufferEXT ( GL_FRAMEBUFFER_EXT, fbo );
glBindTexture ( GL_TEXTURE_2D_ARRAY_EXT, 0 );

//Attach the 3D texture which contains 2 Layers
//to the FBO
glFramebufferTextureEXT ( GL_FRAMEBUFFER_EXT,
  GL_COLOR_ATTACHMENT0_EXT, colorTex, 0 );
...
```

5.1 Efficient Multiple Dynamic Light Shadows

We use Layered Rendering to compute all N shadow maps corresponding to N lights in a single pass. Earlier methods using the Shadow Maps needed N passes for computation of N Depth/Shadow maps. Layered Rendering allows an application to bind an entire complex texture to a framebuffer object, and render primitives to arbitrary layers computed at run time. The layer to render to is specified by writing to the in-built variable *gl_layer*.

Pass 1 : Pass the corresponding view & projection matrices for each light along with the common model matrix of the scene to the Geometry Shader. Each primitive is routed to N layers after multiplying with i^{th} view and projection matrices corresponding to i^{th} Light source and the common Model Matrix, and the Depth Map is saved at i^{th} Layer of the array of 2D textures.

```
...
gl_Layer = lightid;
for (int ii = 0; ii < 3; ii++)
{
    pos = gl_TextureMatrix[0] * gl_PositionIn[ii];

    pos = gl_TextureMatrix[1] * pos;
    gl_Position = gl_TextureMatrix[2] * pos;
    EmitVertex();
}
EndPrimitive();
...
```

Thus, for each incoming triangle above code directs a copy of the triangle to layer number *lightid* by setting *gl_layer* to *lightid*. Texture matrices are used here to store various modelview and projection matrices corresponding to different lights.

Pass 2 : Back-project each pixel in the pixel shader to each of the light's frustum comparing the depth of that particular pixel from light's points of view. We get a boolean answer on whether the pixel in under shadow or not with respect to a particular light source.

```
...
for ( i = -p; i<=p ; i+=1.0 )
    for ( j = -q; j<=q ; j+=1.0 )
    {
        //Light i
        t.x = coordPos0.x + i/TSIZE;
        t.y = coordPos0.y + j/TSIZE;
        t.z = 0;
        depth = texture2DArray( sampler, t ).r;
        depthsqr = depth * depth;
        m01 += depth / 25.0;
        m02 += depthsqr / 25.0;
    }
//sampler is the 2D array texture

//Above is performed for each light computing mean sigma
//of depth values over a window of pxq (soft-shadow
//using Variance Shadow Maps technique).
...
...
//Testing current pixel for shadow from Light i
if ( coordPos0.z >  m01 && coordPos0.x <= 1.0 &&
     coordPos0.x >= 0.0 && coordPos0.y <= 1.0 &&
     coordPos0.y >= 0.0 )
{
     sigma02 = m02 - ( m01 * m01 );
     prob = sigma02 / ( sigma02 + ( (coordPos0.z - m01)
       * (coordPos0.z - m01) ) );
     color = color * prob;
}
...
```

With N comparisons we get N hard shadows corresponding to N light sources in 2 passes. One can use any image based technique to incorporate soft shadows, we have used Variance Shadow Maps [Donnelly and Lauritzen 2006] for the same.

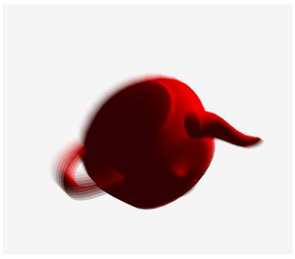

Figure 11: *A rotating Teapot with motion blur.*

5.2 Motion Blur using Layered Rendering

Motion blur is simulated using the accumulation buffer. Accumulation buffers provide a higher precision buffer as compared to frame buffer for adding different images with varying weight. A blended image comprised in accumulation buffer with *n* number of images takes *n + 1* passes. Thus, the number of passes is proportional to the quality of blur needed.

As described with multiple shadow lighting, we can transform this problem too into a two pass algorithm using the Layered Rendering. This example shows how layers can be used to simulate motion blur by rendering to different layers each having a different modelview and optionally projection matrix suited for generating object space blur (Figure 11). We used *m* different *modelview matrices* in order to convey *m* previous positions of the object to the geometry shader using the *Texture matrices* available (32 on Nvidia 8800 GTX).

```
coord.xy = gl_TexCoord[0].xy;
for ( float i=1.0; i<NLAYERS; i+=1.0 )
{
     coord.z = i;
     color += (NLAYERS-i) * texture2DArray(samp,coord);
}
//Handling the current draw position here
coord.z = 0;
color += texture2DArray(samp,coord);

//Making Alpha 1
color.rgba /= color.a;
```

Above code shows how we put together the various framebuffers stored in the first pass in a 2D array texture.

6 Transform Feedback

Stream-Output/Transform-Feedback is a new feature in the rendering pipeline to stream data out in the middle of the pipeline to the memory (Figure 6). This gives the facility of not going through the whole pipeline to process stream data of the object if it is not to be rendered. Also, objects can be rendered and some information, on the fly, can be recorded working in the same pipeline. With this, it is possible to change the state of the object continuously, modifying the original data on the GPU itself, without any interference from the CPU.

6.1 Physics of point objects

Simple physics calculation of objects can be done with only the previous physical state of the object. Instantaneous position and velocity are enough for simple physics simulation of point objects. At any moment, the position and velocity can be updated from the previous state and external forces giving them acceleration. Our GPU-based terrain representation (Section 4) allows quick physics simulation on using the transform feedback feature of the new GPUs. The CPU renders the objects for interaction, such as a bunch of balls released over a terrain.

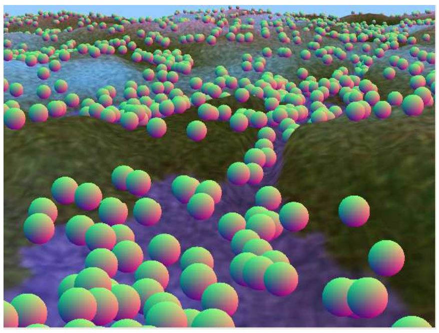

Figure 12: *Screenshot from Shader Model 4 Terrain System. Physics of balls interacting with the terrain*

The CPU triggers a buffer object containing point primitives for each ball with vertex as its initial position and texture coordinate as its initial velocity. The vertex shader computes its next position and velocity according to necessary acceleration (e.g. gravitation) which is determined by its current position and velocity. Vertex shader also accesses the terrain data stored as textures to fetch the height at the object's position. If the object's height is less than that of terrain's, then there is a collision with the terrain. We reflect the velocity vector of the object with respect to the terrain's normal at that position. The changed position and velocity is recorded in another buffer object by the vertex shader to use in the next iteration with the help of transform feedback; the rest of the pipeline is discarded. By doing ping pong between the two buffer objects, the positions and velocities get updated every frame. One of the buffer objects is then used to render an object at its location. We were able to trigger up to 4000 balls (objects) to interact with the terrain, at interactive framerates (21). The performance drops linearly with the increase in number of objects (Graph **??**).

```
glGenBuffersARB ( 1, &tfvbo[0] );
glBindBufferARB ( GL_ARRAY_BUFFER_ARB, tfvbo[0] );
glBufferDataARB ( GL_ARRAY_BUFFER_ARB,
  count*4*sizeof(float), initdata, GL_STATIC_DRAW_ARB );
```

Above code demonstrates how an array buffer is declared for use of storing the streamed output-ed data.

```
glActiveVaryingNV( shader, "varpos\0" );
glActiveVaryingNV( shader, "velocity\0" );

glLinkProgramARB(shader);

loc[0] = glGetVaryingLocationNV ( shader, "varpos\0" );
loc[1] = glGetVaryingLocationNV ( shader, "velocity\0" );
```

Getting a handle to variables which will be recorded during transform feedback.

```
glBindBufferRangeNV ( GL_TRANSFORM_FEEDBACK_BUFFER_NV,
  0, tfvbo[1-current_buffer], 0, 4*count*sizeof(float) );
...
...
glBeginTransformFeedbackNV( GL_LINES );

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_TEXTURE_COORD_ARRAY );

glBindBufferARB( GL_ARRAY_BUFFER_ARB,
  tfvbo[current_buffer] );
glVertexPointer( 4, GL_FLOAT, 0, NULL );
glBindBufferARB ( GL_ARRAY_BUFFER_ARB,
  ttfvbo[current_buffer] );
glTexCoordPointer( 4, GL_FLOAT, 0, NULL );

glDrawArrays( GL_POINTS, 0, 1 );

glDisableClientState( GL_TEXTURE_COORD_ARRAY );
glDisableClientState( GL_VERTEX_ARRAY );

glEndTransformFeedbackNV();
```

Code snippet for rendering using VBOs with transform feedback enabled. The input primitives are set as GL_LINES which should match the Geometry shader input set earlier.

```
...
varpos = gl_PositionIn[0];
velocity = gl_TexCoordIn[0][0];

varpos.x +=  0.01;
velocity.x -= 0.01;
...
```

Geometry Shader code where the variables of our choice are recorded with transform feedback.

The bottleneck of transfering updated physics state to the GPU from CPU for physics is overcome by keeping CPU oblivious to the physics state. Complete physics can be performed on the GPU using the transform feedback, thus cutting on the precious CPU-GPU bandwidth. A major advantage of performing physics using the transform feedback on the GPU is that its highle scalable. We have tested the physics for point objects for high numbers and the results we get are very convincing. We find out that the computation load grows linearly with the increase in balls (Figure **??**).

We use array textures to represent the GPU cache of the terrain to facilitate the interaction of objects with the terrain as mentioned in Section 4. This makes it possible to use the layer ID of an individual block as a variable. Thus, an appropriate block of the cache can be selected based on the external object's (x, y) location. Conventional textures are statically bound and their IDs cannot be variable.

References

ASIRVATHAM, A., AND HOPPE, H. 2005. Terrain rendering using gpu-based geometry clipmaps. *GPU Gems 2*, 46–53.

BHATTACHARJEE, S., PATIDAR, S., AND NARAYANAN, P. J. 2007. Technical report on gpu-resident terrains. Tech. rep.

BLYTHE, D. 2006. The direct3d 10 system. *ACM Transactions of Graphics 25*, 3, 724–734.

CROW, F. C. 1977. Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 242–248.

DONNELLY, W., AND LAURITZEN, A. 2006. Variance shadow maps. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, ACM Press, New York, NY, USA, 161–165.

GOVINDARAJU, N. K., LARSEN, S., GRAY, J., AND MANOCHA, D. 2006. Memory—a memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM Press, New York, NY, USA, 89.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. *ACM Trans. Graph. 21*, 3, 355–361.

HARRIS, W., 2005. bit-tech.net, a bluffer's guide to shader models.

LOSASSO, F., AND HOPPE, H. 2004. Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Trans. Graph. 23*, 3, 769–776.

NVIDIA, 2004. Improve batching using texture atlases, nvidia sdk white paper.

WAGNER, D. 2004. Terrain geomorphing in the vertex shader. *ShaderX2, Shader Programming Tips and Tricks with DirectX 9, Wordware Publishing*.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. 270–274.