CRC Press
Taylor & Francis Group
AN A K PETERS BOOK

GPU Pro 360

Guide to 3D Engine Design

Edited by **Wolfgang Engel**

GPU Pro 360

Guide to 3D Engine Design

GPU Pro 360

Guide to 3D Engine Design

Edited by Wolfgang Engel

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper

International Standard Book Number-13: 978-0-8153-9075-6 (Paperback)
International Standard Book Number-13: 978-0-8153-9079-4 (Hardback)

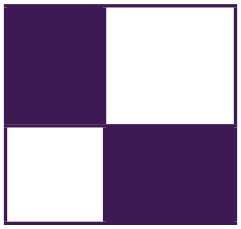
Library of Congress Cataloging-in-Publication Data

Names: Engel, Wolfgang F., editor.
Title: GPU pro 360 guide to 3D engine design / [edited by] Wolfgang Engel.
Description: First edition. | Boca Raton, FL : CRC Press/Taylor & Francis Group, 2018. | Includes bibliographical references and index.
Identifiers: LCCN 2018020471| ISBN 9780815390756 (pbk. : acid-free paper) |
ISBN 9780815390794 (hardback : acid-free paper)
Subjects: LCSH: Computer graphics. | Graphics processing units--Programming.
| Graphics processing units--Design and construction. | Three-dimensional modeling.
Classification: LCC T385 .G688778 2018 | DDC 006.6/93--dc23
LC record available at https://lccn.loc.gov/2018020471

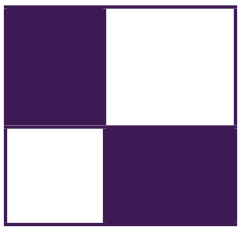
Visit the eResources: www.crcpress.com/9780815390756

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com



Contents



Introduction

This book targets the design of a renderer. A renderer is a very complex software module that requires attention to a lot of details. The requirements and attention also vary greatly on different hardware platforms. The chapters here cover various aspects of engine design, such as quality and optimization, in addition to high-level architecture.

The chapter "Multi-Fragment Effects on the GPU Using Bucket Sort" covers a technique on how to render order-independent transparency by utilizing a bucket sort system. Traditionally, pixels or fragments are processed in depth order rather than rasterization order and modern GPUs are optimized to capture the nearest and furthest fragment per pixel in each geometry pass. Depth peeling offers a simple and robust solution by peeling off one layer per pass, but rasterizing depth data multiple times leads to performance bottlenecks. Newer approaches like the K-buffer approach capture fragments in a single pass but suffer from read-modify-write (RMW hazards). This chapter presents a method that utilizes a bucket array per pixel that is allocated using MRT as the storage. It is more efficient than classical depth peeling while offering good visual results.

The next chapter, "Parallelized Light Pre-Pass Rendering with the Cell Broadband Engine," demonstrates the efficient implementation of a light pre-pass / deferred lighting engine on the PS3 platform. Distributing the workload of rendering many lights over the SPE and GPU requires some intricate knowledge of the platform and software engineering skills that are covered in the chapter.

In Stephen Hill and Daniel Collin's chapter "Practical, Dynamic Visibility for Games," the authors introduce methods for determining per-object visibility, taking into account occlusion by other objects. The chapter provides invaluable and inspiring experience from published AAA titles, showing excellent gains that are otherwise lost without this system.

Next, Eric Penner presents "Shader Amortization Using Pixel Quad Message Passing." In this chapter, he analyzes one particular aspect of modern programmable hardware: the pixel derivative instructions and pixel quad rasterization. The chapter identifies a new level at which optimizations can be performed, and applies this method to achieve results such as 4×4 percentage closer filtering (PCF) using only one texture fetch, and 2×2 bilateral up-sampling using only one or two texture fetches.

On the topic of crowd rendering, the chapter "A Rendering Pipeline for Real-Time Crowds," by Benjamín Hernández and Isaac Rudomin, describes a detailed system for simulating and rendering large numbers of different characters on the GPU, making use of optimizations such as culling and LOD-selection to improve performance of the system.

Pascal Gautron, Jean-Eudes Marvie, and Gaël Sourimant present us with the chapter, "Z^3 Culling," in which the authors suggest a novel method to optimize depth testing over the Z-buffer algorithm. The new technique adds two "depth buffers" to keep the early-Z culling optimization even on objects drawn with states that prevent early-Z culling (such as alpha testing).

Next, Dzmitry Malyshau brings his experience of designing a quaternion-based 3D engine in his chapter, "Quaternion-Based Rendering Pipeline." Malyshau shows the benefits of using quaternions in place of transformation matrices in various steps of the rendering pipeline based on his experience of a real-world 3D-engine implementation.

In the chapter, "Implementing a Directionally Adaptive Edge AA Filter Using DirectX 11," Matthew Johnson improves upon the box antialiasing filter using a postprocessing technique that calculates a best fit gradient line along the direction of candidate primitive edges to construct a filter that gives a better representation of edge information in the scene, and thus higher quality antialiased edges.

Donald Revie describes the high-level architecture of a 3D engine in the chapter "Designing a Data-Driven Renderer." The design aims to bridge the gap between the logical simulation at the core of most game engines and the strictly ordered stream of commands required to render a frame through a graphics API. The solution focuses on providing a flexible data-driven foundation on which to build a rendering pipeline, making minimal assumptions about the exact rendering style used.

Next, Donald Revie brings his experience of engine design in his chapter "An Aspect-Based Engine Architecture." Aspect-based engines apply the principles of component design and object-oriented programming (OOP) on an engine level by constructing the engine using modules. Such architecture is well suited to small or distributed teams who cannot afford to establish a dedicated structure to design and manage all the elements of their engine but would still like to take advantage of the benefits that developing their own technology provides. The highly modular nature allows for changes in development direction or the accommodation of multiple projects with widely varying requirements.

In the chapter "Kinect Programming with Direct3D 11," Jason Zink provides a walkthrough into this emerging technology by explaining the hardware and software aspects of the Kinect device. The chapter seeks to provide the theoretical underpinnings needed to use the visual and skeletal data streams of the Kinect, and it also provides practical methods for processing and using this data with the Direct3D 11 API. In addition, it explores how this data can be used in real-time rendering scenarios to provide novel interaction systems.

Homam Bahnassi and Wessam Bahnassi present a description of a full pipeline for implementing structural damage to characters and other environmental objects in the chapter "A Pipeline for Authored Structural Damage." The chapter covers details for a full pipeline from mesh authoring to displaying pixels on the screen, with qualities including artist-friendliness, efficiency, and flexibility.

Next, Peter Sikachev, Vladimir Egorov, and Sergey Makeev share their experience using quaternions in an MMORPG game engine. Their chapter, "Quaternions Revisited," illustrates the use of quaternions for multiple purposes in order to replace bulky 3×3 rotation and tangent space matrices throughout the entire engine, most notably affecting aspects such as normal mapping, generic transforms, instancing, skinning, and morph targets. The chapter shows the performance and memory savings attributable to the authors' findings.

Fabrice Robinet, Rémi Arnaud, Tony Parisi, and Patrick Cozzi present the chapter "glTF: Designing an Open-Standard Runtime Asset Format." This chapter introduces work by the COLLADA Working Group in the Khronos Group to provide a bridge between interchange asset formats and the OpenGL-based runtime graphics APIs (e.g., WebGL and OpenGL ES). The design of the glTF open-standard transmission-format is described, along with open-source content pipeline tools involved in converting COLLADA to glTF and REST-based cloud services.

Bartosz Chodorowski and Wojciech Sterna present the chapter "Managing Transformations in Hierarchy," which provides a study on this basic 3D engine component. In addition to presenting the theory, it describes and addresses some of the issues found in common implementations of the transformation hierarchy system. It also describes how to achieve some useful operations within this system such as re-parenting nodes and global positioning.

Holger Gruen examines the benefits of a block-wise linear memory layout for binary 3D grids in the chapter "Block-Wise Linear Binary Grids for Fast Ray-Casting Operations." This memory layout allows mapping a number of volumetric intersection algorithms to binary `AND` operations. Bulk-testing a subportion of the voxel grid against a volumetric stencil becomes possible. The chapter presents various use cases for this memory layout optimization.

Michael Delva, Julien Hamaide, and Ramses Ladlani present the chapter "Semantic-Based Shader Generation Using Shader Shaker." This chapter offers one solution for developing and efficiently maintaining shader permutations across multiple target platforms. The proposed technique produces shaders automatically from a set of handwritten code fragments, each responsible for a single feature. This particular version of the proven divide-and-conquer methodology differs in the way the fragments are being linked together by using a path-finding algorithm to compute a complete data flow through shader fragments from the initial vertex attributes to the final pixel shader output.

Shannon Woods, Nicolas Capens, Jamie Madill, and Geoff Lang present the chapter "ANGLE: Bringing OpenGL ES to the Desktop." ANGLE is a portable,

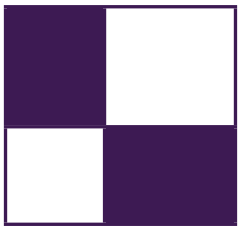
open-source, hardware-accelerated implementation of OpenGL ES 2.0 used by software like Google Chrome. The chapter provides a close insight on the Direct3D 11 backend implementation of ANGLE along with how certain challenges were handled, in addition to recommended practices for application developers using ANGLE.

Homam and Wessam Bahnassi describe a new real-time particle simulation method that works by capturing simulation results from DCC tools and then replaying them in real time on the GPU at a low cost while maintaining the flexibility of adding interactive elements to those simulations. Their technique "Interactive Cinematic Particles" has been applied successfully in the game *Hyper Void*, which runs at 60 fps even on the Playstation 3 console.

Krzysztof Narkowicz presents the chapter "Real-Time BC6H Compression on GPU." The chapter describes a simple real-time BC6H compression algorithm, one which can be implemented on GPU entirely with practical performance figures. Such a technique can be very useful for optimizing rendering of dynamic HDR textures such as environment cubemaps.

The next chapter by Gustavo Bastos Nunes is "A 3D Visualization Tool Used for Test Automation in the Forza Series." The tool introduced automatically analyzes a mesh for bad holes and normal data and gives the manual tester an easy semantic view of what are likely to be bugs and what are by-design data. The tool was used during the entire production cycle of *Forza Motorsport 5* and *Forza: Horizon 2* by Turn 10 Studios and Playground Games, saving several hundred hours of manual testing and increasing trust in shipping the game with collision meshes in a perfect state.

Finally, Takahiro Harada presents the chapter "Semi-Static Load Balancing for Low-Latency Ray Tracing on Heterogeneous Multiple GPUs," which describes a low-latency ray tracing system for multiple GPUs with nonuniform compute powers. To realize the goal, a semi-static load balancing method is proposed that uses rendering statistics of the previous frame to compute work distribution for the next frame. The proposed method does not assume uniform sampling density on the framebuffer, thus it is applicable for a problem with an irregular sampling pattern. The method is not only applicable for a multi-GPU environment, but it can be used to distribute compute workload on GPUs and a CPU as well.



Web Materials

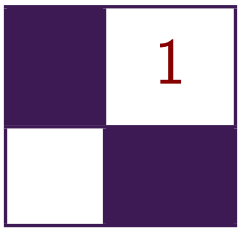
Example programs and source code to accompany some of the chapters are available on the CRC Press website: go to https://www.crcpress.com/9780815390756 and click on the "Downloads" tab.

The directory structure follows the book structure by using the chapter numbers as the name of the subdirectory.

General System Requirements

The material presented in this book was originally published between 2010 and 2016, and the most recent developments have the following system requirements:

- The DirectX June 2010 SDK (the latest SDK is installed with Visual Studio 2012).
- DirectX 11 or DirectX 12 capable GPUs are required to run the examples. The chapter will mention the exact requirement.
- The OS should be Microsoft Windows 10, following the requirement of DirectX 11 or 12 capable GPUs.
- Visual Studio C++ 2012 (some examples might require older versions).
- 2GB RAM or more.
- The latest GPU driver.



Multi-Fragment Effects on the GPU Using Bucket Sort

Meng-Cheng Huang, Fang Liu, Xue-Hui Liu, and En-Hua Wu

1.1 Introduction

Efficient rendering of multi-fragment effects has long been a great challenge in computer graphics, which always require to process fragments in depth order rather than rasterization order. The major problem is that modern GPUs are optimized only to capture the nearest or furthest fragment per pixel each geometry pass. The classical depth peeling algorithm [Mammen 89, Everitt 01] provides a simple but robust solution by peeling off one layer per pass, but multi-rasterizations will lead to performance bottleneck for large-scale scene with high complexity. The k-buffer [Bavoil et al. 07, Liu et al. 06] captures k fragments in a single pass but suffers from serious read-modify-write(RMW) hazards.

This chapter presents a fast approximation method for efficient rendering of multi-fragment effects via bucket sort on GPU. In particular, a bucket array of size K is allocated per pixel location using MRT as storage, and the depth range of each pixel is consequently divided into K subintervals. During rasterization, fragments within the kth ($k = 0, 1, \cdots, K-1$) subinterval will be routed to the kth bucket by a bucket sort. Collisions will happen when multiple fragments are routed to the same bucket, which can be alleviated by multi-pass approach or an adaptive scheme. Our algorithm shows great speedup to the classical depth peeling with visually faithful results, especially for large scenes with high complexity.

1.2 Design of Bucket Array

The bucket array can be constructed as a fixed size buffer per pixel location in GPU memory, thus the MRT buffers turn out to be a natural candidate. Since modern GPUs can afford at most eight MRTs with internal pixel format of `GL_RGBA32F_ARB`, the size of our bucket array can reach up to 32, which is often enough for most common applications.

The default `REPLACE` blending of the MRTs will introduce two problems. First, when multiple fragments are trying to update the bucket array on the same pixel location concurrently, the number of the operations on that location and the order in which they occur is undefined, and only one of them is guaranteed to succeed. Thus they will produce unpredictable results under concurrent writes. Second, modern GPUs have not yet supported independent update of arbitrary channels of MRT buffers. The update of a specific channel of the MRT buffers will result in all the remaining channels being overwritten by the default value zero simultaneously. As a result, the whole bucket array will hold at most one depth value at any time.

Fortunately, these problems can be solved via the 32-bit floating-point MAX/MIN blending operation, which is available on recent commodity NVIDIA GeForce 8 or its ATI equivalents. Take the MAX blending operation as an example, which performs a comparison between the source and the destination values of each channel of MRTs, and keeps the greater one of each pair. This atomic operation guarantees all the read, modify, write operations to the same pixel location will be serialized and performed without interference from each other, thus completely avoiding the first problem of RMW hazards.

The second problem can be solved by initializing each bucket of the bucket array to zero. When updating a certain bucket, if the original value in the bucket is zero, the update will always succeed since the normalized depth values are always greater than or equal to zero; otherwise, the greater one will survive the comparison. As for other buckets, we implicitly update them simultaneously by the default value zero so that their original values are always greater and can be kept unchanged. When multiple fragments are routed to the same bucket, i.e., a collision happens, the MAX blending operation assures that the maximum depth value will win all the tests and finally stay in the bucket. MIN blending is performed in a similar way except initializing each bucket and explicitly updating the other buckets by one. The MAX/MIN blending operation enables us to update a specific bucket independently, and guarantees correct results free of RMW hazards. Since the default update value for each channel of the MRT is zero, we prefer to utilize MAX blending in our implementation for simplicity.

1.3 The Algorithm

The depth value of each fragment is normalized into a range [0,1], but for most pixels, the geometry only occupies a small subrange. Thus a bounding box or a coarse visual hull can be first rendered to approximate the depth range $[z\text{Near}, z\text{Far}]$ per pixel in the same way as dual depth peeling [Bavoil and Myers 08]. During rasterization, the consecutive buckets per pixel are bind into 16 pairs and the depth range are divided into 16 corresponding subintervals uniformly. We then perform the dual depth peeling within each subinterval concurrently. For a fragment with depth value d_f, the corresponding bucket pair index k can be computed as follows:

$$k = \text{floor}\left(\frac{16 \times (d_f - z\text{Near})}{z\text{Far} - z\text{Near}}\right).$$

Then the kth pair of buckets will be updated by $(1 - d_f, d_f)$ and the rest pairs by $(0, 0)$. When the first geometry pass is over, the minimum and maximum depth values within the kth subinterval can be obtained from the kth pair of buckets, i.e.,

$$\text{dmin}_k^1 = 1 - \max_{d_f \in [d_k, d_{k+1})}(1 - d_f), \quad \text{dmax}_k^1 = \max_{d_f \in [d_k, d_{k+1})}(d_f).$$

It is obvious that these fragments in the consecutive depth intervals are in correct depth ordering:

$$\text{dmin}_0^1 \le \text{dmax}_0^1 \le \text{dmin}_1^1 \le \text{dmax}_1^1 \le \cdots \le \text{dmin}_{15}^1 \le \text{dmax}_{15}^1.$$

If there is no fragment within the kth subinterval, both dmax_k^1 and dmin_k^1 will remain the initial value 0 and can be omitted. While if there is only one fragment within the kth subinterval, dmax_k^1 and dmin_k^1 will be equal and one of them can be eliminated. In a following fullscreen pass, the bucket array will be sequentially accessed as eight input textures to retrieve the sorted fragments for post-processing.

For applications that need other fragment attributes, taking order independent transparency as an example, we can pack the RGBA8 color into a 32-bit positive floating-point using the Cg function *pack_4ubyte*. The alpha channel will be halved and mapped to the highest byte to ensure the positivity of the packed floating-point. We then divide the depth range into 32 subintervals corresponding to the 32 buckets and capture the packed colors instead of the depth values in a similar way. In post-processing, we can unpack the floating-point colors to RGBA8 and double the alpha channel for blending.

1.4 Multi-Pass Approach

The algorithm turns out to be a good approximation for uniformly distributed scenes with few collisions. But for non-uniform ones, collisions will happen more frequently especially on the silhouette or details of the model with noticeable artifacts. The algorithm can be extended to a multi-pass approach for better results. In the second geometry pass, we allocate a new bucket array for each pixel and the bucket array captured in the first pass will be taken as eight input textures. For a fragment within the kth subinterval, if its depth value d_f satisfies condition $d_f \geq \mathrm{dmax}_k^1$ or $d_f \leq \mathrm{dmin}_k^1$, it must have been captured in the previous pass, thus can be simply discarded. When the second pass is over, the second minimal and maximum depth values dmin_k^2 and dmax_k^2 in the kth subinterval can be retrieved from the kth pair of buckets similarly. The depth values captured in these two passes are naturally in correct ordering:

$$\mathrm{dmin}_0^1 \leq \mathrm{dmin}_0^2 \leq \mathrm{dmax}_0^2 \leq \mathrm{dmax}_0^1 \leq \mathrm{dmin}_1^1 \leq \mathrm{dmin}_1^2 \leq \mathrm{dmax}_1^2,$$

$$\leq \mathrm{dmax}_1^1 \leq \cdots \leq \mathrm{dmin}_{15}^1 \leq \mathrm{dmin}_{15}^2 \leq \mathrm{dmax}_{15}^2 \leq \mathrm{dmax}_{15}^1.$$

During post-processing, both bucket arrays can be passed to the pixel shader as input textures and accessed for rendering of multi-fragment effects.

Theoretically, we can obtain accurate results by enabling the occlusion query and looping in the same way until all the fragments have been captured. However, the sparse layout of depth values in the bucket arrays will lead to memory exhaustion especially for non-uniform scenes and high screen resolutions. Artifacts may also arise due to the inconsistency between the packed attribute ordering and the correct depth ordering. We instead propose a more robust scheme to alleviate these problems at the cost of an additional geometry pass, namely adaptive bucket depth peeling. The details will be described as follows.

1.5 The Adaptive Scheme

The uniform division of the depth range may result in some buckets overloaded while the rest idle for non-uniform scenes. Ideally, we prefer to adapt the division of subintervals to the distribution of the fragments per pixel, so that there is only one fragment falling into each subinterval. The one-to-one correspondence between fragments and subintervals will assure only one fragment for each bucket, thus can avoid the collisions.

Inspired by the image histogram equalization, we define a depth histogram as an auxiliary array with each entry indicating the number of fragments falling into the corresponding depth subinterval, thus is a probability distribution of the geometry. We allocate eight MRT buffers with pixel format `GL_RGBA32UI_EXT`

as our depth histogram. Considering each channel of the MRT as a vector of 32 bits, the depth histogram can be cast to a bit array of size 4*8*32=1024, with each bit as a binary counter for fragments. Meanwhile, the depth range is divided into 1024 corresponding subintervals: $[d_k, d_{k+1}), d_k = z\text{Near} + \frac{k}{1024}(z\text{Far} - z\text{Near}), k = 0, 1, \cdots, 1023$. The depth range is always on a magnitude of 10^{-1}, so the subintervals will be on a magnitude of 10^{-4}, which are often small enough to distinguish almost any two close layers. As a result, there is at most one fragment within each subinterval on most occasions, thus a binary counter for each entry of the depth histogram will be sufficient most of the time.

Similarly, we begin by approximating the depth range per pixel by rendering the bounding box of the scene in an initial pass. In the first geometry pass, an incoming fragment within the kth subinterval will set the kth bit of the depth histogram to one using the OpenGL's 32-bit logic operation `GR_OR`. After the first pass, each bit of the histogram will indicate the presence of fragments in that subinterval or not. A simplified example with depth complexity $N = 8$ (the maximum number of layers of the scene at all viewing angles) is

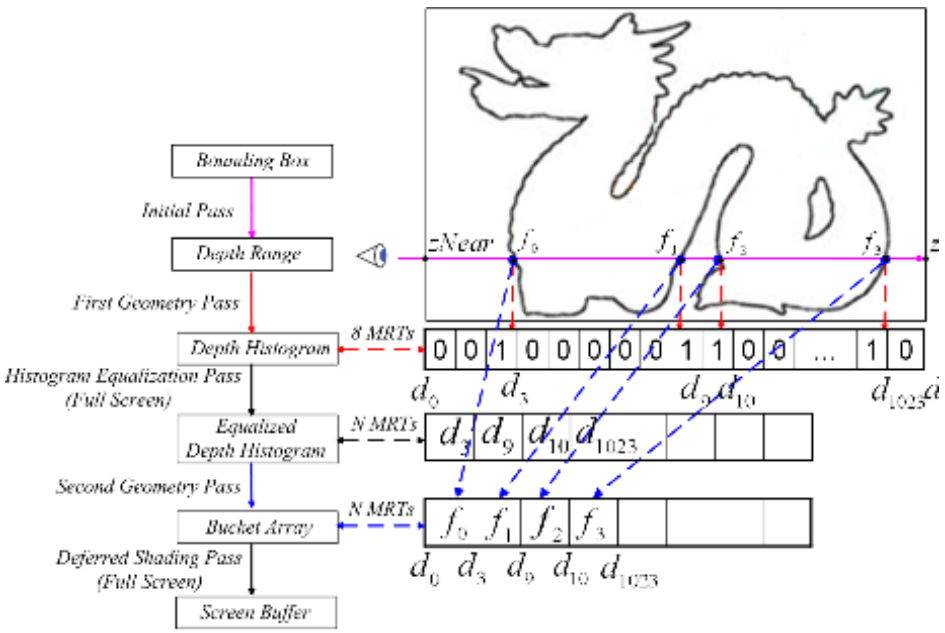


Figure 1.1. An example of adaptive bucket depth peeling. The red arrows indicate the operations in the first geometry pass and the blue arrows indicate the operations in the second geometry pass.

```
void main(  float4  wpos : WPOS,
            uniform samplerRECT depthRange,

            //Output histogram as eight MRTs.
            out unsigned int4  color0 : COLOR0,
            out unsigned int4  color1 : COLOR1,
            ......
            out unsigned int4  color7 : COLOR7 )
{
  float z = wpos.z;
  float4 range = texRECT(depthRange, wpos.xy);

  float zNear = 1 - range.x;
  float zFar = range.y;

  int k = floor( 1024 * ( z-zNear)/(zFar-zNear) );
  int i = k >> 5;
  int j = k & 0x1F;

  unsigned int SetBit = 0x80000000 >> j;

  if(i==0) color0 = unsigned int4(SetBit,0,0,0);
  else if(i==1) color0 = unsigned int4(0,SetBit,0,0);
  else if(i==2) color0 = unsigned int4(0,0,SetBit,0);
  ......
  else if(i==30) color7 = unsigned int4(0,0,SetBit,0);
  else color7 = unsigned int4(0,0,0,SetBit);
}
```

Listing 1.1. The pixel shader in the first geometry pass.

shown in Figure 1.1. Suppose at a certain pixel location, the eye ray intersects the scene generating four fragments $f_0 - f_3$ within four different subintervals $[d_2, d_3], [d_8, d_9], [d_9, d_{10}], [d_{1022}, d_{1023}]$. They will set the 3rd, 9th, 10th, and the 1023rd bit of the depth histogram to 1 in the first geometry pass. The code snippet Listing 1.1 shows the pixel shader in the first geometry pass.

The depth histogram is equalized in a following fullscreen pass. For scenes with depth complexity N less than 32, the histogram is passed into the pixel shader as eight input textures, and new floating-point MRT buffers with N channels will be allocated as an equalized histogram for output. We can consecutively obtain the jth bit of the ith $(i, j = 0, 1, 2, \cdots, 31)$ channel of the input depth histogram. If the bit is zero, it means that there is no fragment falling into the kth $(k = i*32+j)$ depth subinterval, thus can be simply skipped over; otherwise, there is at least one fragment within that subinterval, so we store the corresponding upper bound d_{k+1} consecutively into the equalized histogram for output. As for the example in Figure 1.1, two MRT buffers with eight channels will be al-

located as the equalized histogram, and the upper bounds d_3, d_9, d_{10}, and d_{1023} will be stored into it in the equalization pass. The code snippet Listing 1.2 shows the pixel shader in the histogram equalization pass.

```
void main(  float4  wpos : WPOS,
            uniform samplerRECT depthRange,
            //Input histogram as eight textures.
            usamplerRECT fbcolor0,
            usamplerRECT fbcolor1,
            ......
            usamplerRECT fbcolor7,
            //Output equalized histogram as eight MRTs.
            out float4 color0 : COLOR0,
            out float4 color1 : COLOR1,
            ......
            out float4 color7 : COLOR7 )
{
  float4 range = texRECT(depthRange, wpos.xy);
  float zFar = range.y;  if( zFar == 0 ) discard;
  float zNear = 1 - range.x;

  unsigned int4 fb0 =  texRECT(fbcolor0, wpos.xy);
  unsigned int4 fb1 =  texRECT(fbcolor0, wpos.xy);
  ......
  unsigned int4 fb7 =  texRECT(fbcolor7, wpos.xy);

  // Discard pixels that are not rendered.
  if( any( fb0|fb1|fb2|fb3|fb4|fb5|fb6|fb7 ) == 0 ) discard;

  unsigned int Histogram[32];
  Histogram[0]=fb0.x; Histogram[1]=fb0.y;
  Histogram[2]=fb0.z; Histogram[3]=fb0.w;
  ......
  Histogram[30]=fb7.z; Histogram[31]=fb7.w;

  float EquHis[32]; // Equalized histogram
  float coeff = (zFar - zNear) / 1024.0;
  int HisIndex = 1, EquHisIndex= 0;

  for(int i = 0; i < 32; i++, HisIndex += 32)
  {
    unsigned int remainded = Histogram[i];
    // End the inner loop when the remained bits are all zero.
    for(int k = HisIndex; remainded != 0; k++, remainded <<= 1)
    {
      if(remainded >= 0x80000000)
      {
        // The $k$th bit of the histogram has been set to one,
        // so store the upper bound of the $k$th subinterval.
        EquHis[EquHisIndex++] = k * coeff + zNear;
      }
    }
```

```
    }
    color0 = float4(EquHis[0],EquHis[1],EquHis[2],EquHis[3]);
    color1 = float4(EquHis[4],EquHis[5],EquHis[6],EquHis[7]);
    ......
    color7 = float4(EquHis[28],EquHis[29],EquHis[30],EquHis[31]);
}
```

Listing 1.2. The pixel shader of the histogram equalization pass.

We perform the bucket sort in the second geometry pass. The equalized histogram is passed to the pixel shader as input textures and a new bucket array of the same size N is allocated as output for each pixel. The upper bounds in the input equalized histogram will redivide the depth range into non-uniform subintervals with almost one-to-one correspondence between fragments and subintervals. As a result, there will be only one fragment falling into each bucket on most occasions; thus collisions can be reduced substantially. During rasterization, each incoming fragment with a depth value d_f will search the input equalized histogram (denoted as EquHis for short). If it belongs to the kth subinterval, i.e., it satisfies conditions $d_f \geq \text{EquHis}[k-1]$ and $d_f < \text{EquHis}[k]$, it will be routed to the kth bucket. In the end, the fragments are consecutively stored in the output bucket array, so our adaptive scheme will be memory efficient. The bucket array will then be passed to the fragment shader of a fullscreen deferred shading pass as textures for post-processing. As for our example in Figure 1.1, the upper bounds in the equalized histogram redivide the depth range into 4 subintervals: $[0, d_3), [d_3, d_9), [d_9, d_{10}), [d_{10}, d_{1023}]$. Fragment f_0 is within the first subinterval $[0, d_3)$, so it is routed to the first bucket. Fragment f_1 is within the second subinterval $[d_3, d_9)$, and is routed to the second bucket, and so on. After the second geometry pass, all of the four fragments are stored in the bucket array for further applications.

This adaptive scheme can reduce the collisions substantially, but collisions might still happen when two close layers of the model generate two fragments with a distance less than 10^{-4}, especially on the silhouette or details of the model. These fragments are routed to the same bucket and merged into one layer, thus resulting in artifacts. In practice, we can further reduce collisions by binding the buckets into pairs and performing dual depth peeling within each non-empty subinterval. Theoretically, the multi-pass approach can be resorted to for better results.

For applications that need multiple fragment attributes, the one-to-one correspondence between fragments and subintervals can assure the attributes consistency, so we can bind consecutive buckets into groups and update each group by the attributes simultaneously.

For scenes with more than 32 layers, we can handle the remaining layers by scanning over the remaining part of the histogram in a new fullscreen pass to get another batch of 32 nonzero bits. We then equalize it and pass the equalized histogram to the next geometry pass to route the fragments between layer 32 and 64 into corresponding buckets in the same way, and so on, until all the fragments have been captured.

1.6 Applications

Many multi-fragment effects can benefit from our algorithm and gain high performance in comparison to the previous methods. To demonstrate the results, we took several typical ones as examples. Frame rates are measured at 512×512 resolution on an NVIDIA 8800 GTX graphics card with driver 175.16 and Intel Duo Core 2.4G Hz with 3GB memory.

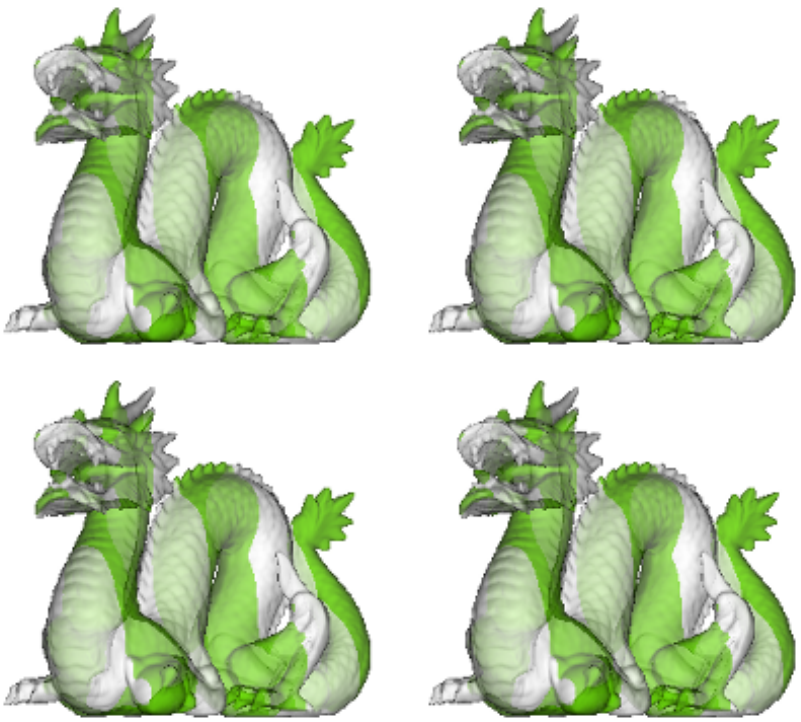


Figure 1.2. Transparent effect on Stanford Dragon (871K triangles). The left top is rendered by BDP (256fps); the right top is by BDP2 (128fps); the left bottom is by ADP (106fps); and the right bottom is the ground truth generated by DP (24fps).

1.6.1 Transparent Effect

Figure 1.2 shows the order independent transparent effect on Stanford Dragon rendered by our bucket depth peeling with a single pass (BDP) and its two-pass extension (BDP2) and the adaptive bucket depth peeling (ADP) in comparison to the classical depth peeling (DP). The differences between the results of our algorithm and the ground truth genertated by DP are visually unnoticeable, so one pass would be a good approximation when performance is more crucial.

1.6.2 Translucent Effect

The translucent effect can be rendered accounting only for absorption and ignoring reflection [NVIDIA 05]. The ambient term I_a can be computed using

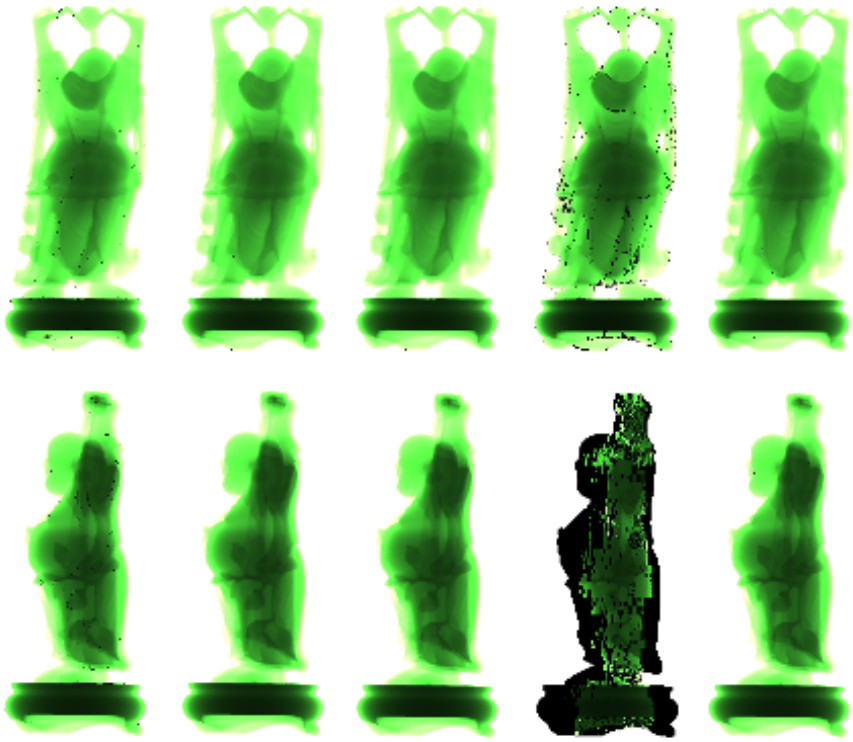


Figure 1.3. Translucent effect on the Buddha model (1,087K triangles). The first column is rendered by BDP (212fps); the second and third are by BDP2 and ADP (106fps); the third is by the k-buffer of 16 layers without modifications (183fps); and the last one is the ground truth generated by DP (20fps).

Beer-Lambert's law: $I_a = \exp(-\sigma_t l)$, where σ is the absorption coefficient and l is the accumulated distance that light travels through the material, which can be approximated by accumulating the thickness between every two successive layers of the model per pixel. As a result, the translucent effect is quite sensitive to errors. Figure 1.3 shows the translucent effect on the Buddha model using different methods. Experimental results show that for k-buffer the RMW hazards are more severe on the side views with more layers, while in contrast, the single pass BDP provides a good approximation and the two-pass approach or the adaptive scheme is preferred for better visual quality.

1.6.3 Fresnel's Effect

Taking into account the attenuation of rays, Schlick's approximation can be used for fast calculation of Fresnel's transmittance of each fragment: $Ft = 1 - (1 - \cos(\theta))^5$. Figure 1.4 shows the results of Fresnel's effect rendered by ADP. In the second geometry pass, we transform the normal into eye space and pack it into a positive floating-point using the Cg function `pack_4byte`. The buckets are bind into pairs and each pair will be updated by the packed normal and the depth value simultaneously. In the deferred shading pass, the ambient term of each pixel can be obtained using Beer-Lambert's law. For a certain pixel, the eye direction can be restored by transforming the fragment position from the screen space back to the the eye space. We then unpack the normal of each fragment and perform a dot product with the eye direction to get the incident angle θ on that

Figure 1.4. Fresnel's effect on the Buddha model (1,087K triangles) rendered by ADP.

surface. In the end, Fresnel's transmittance of each fragment can be computed and multiplied together as the final attenuating factor to the ambient term on that pixel location. The code snippet Listing 1.3 shows the pixel shader for the deferred shading of the Fresnel's effect.

More applications such as constructive solid geometry (CSG), depth of field, shadow maps, refraction, and volume rendering will also benefit from our algorithms greatly in a similar way.

```
// Restore the eye-space position of the fragment from the depth
//  value.
float3 TexToEye(float2 pixelCoord, float eye_z,float2 focusLen)
{
    pixelCoord.xy -= float2(0.5, 0.5);
    pixelCoord.xy /= float2(0.5, 0.5);
    float2 eye_xy = (pixelCoord.xy / focusLen) * eye_z;
    return float3(eye_xy, eye_z);
}

void main(  float4 pixleCoordinate : TEXCOORD0,
            float4 wpos            : WPOS,
            uniform float2 focusLength,//Focus length of camera
            //Input bucket array as eight textures.
            uniform samplerRECT fbcolor0,
            uniform samplerRECT fbcolor1,
            ......
            uniform samplerRECT fbcolor7,
            out float4 color : COLOR)
{
  float4 fb0 = texRECT(fbcolor0, wpos.xy);
  float4 fb1 = texRECT(fbcolor1, wpos.xy);
  ......
  float4 fb7 = texRECT(fbcolor7, wpos.xy);

  unsigned int DepthNormal[32]; //Depth value and packed normal
  DepthNormal[0]=fb0.x; DepthNormal[1]=fb0.y;
  DepthNormal[2]=fb0.z; DepthNormal[3]=fb0.w;
  ......
  DepthNormal[30]=fb7.z; DepthNormal[31]=fb7.w;

  float thickness = 0;
  float x = -1;
  float coeff = 1.0; //The final attenuating factor

  for(int i=0;i<32;i+=2)
  {
    if( DepthNormal[i] > 0 )
    {
      float z = DepthNormal[i];
      thickness += x*z; //Accumulating the thickness
      x = -x;
      //Unpack eye-space normal N.
```

```
        float3 N = normalize(unpack_4byte(DepthNormal[i+1]).xyz);
        //Compute eye-space position P and incident direction I.
        float3 P = TexToEye(pixleCoordinate.xy,z,focusLength);
        float3 I = normalize(P);
        float cosTheta = abs(dot(I,N));  //Incident angle
        coeff *= (1-pow(1.0-cosTheta,5));//Fresnel's transmittance
      }
    }
    if( thickness == 0 ) discard;
    float4 jade = float4(0.14,0.8,0.11,1.0) * 8;
    color = exp(-30*thickness) * jade * coeff;
  }
```

Listing 1.3. The pixel shader for rendering of Fresnel's effect.

1.7 Conclusions

This chapter presents a novel framework of bucket depth peeling, the first linear algorithm for rendering multi-fragment effects via bucket sort on GPU. Experiment results show great speedup to classical depth peeling with faithful results, especially for large-scale scenes with high depth complexity.

The main disadvantages are the approximate nature of the algorithm and the large memory overhead. In the future, we are interested in forming more efficient schemes to reduce collisions further more. In addition, the memory problem might be alleviated by composing the fragments within each bucket per pass, and finally composing all the buckets after done.

Bibliography

[Bavoil and Myers 08] Louis Bavoil and Kevin Myers. "Order Independent Transparency with Dual Depth Peeling." Technical report, NVIDIA Corporation, 2008.

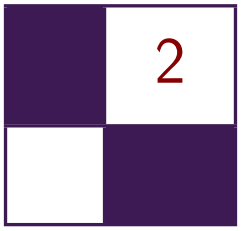
[Bavoil et al. 07] Louis Bavoil, Steven P. Callahan, Aaron Lefohn, Jo ao L. D. Comba, and Cláudio T. Silva. "Multi-Fragment Effects on the GPU Using the k-Buffer." In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pp. 97–104, 2007.

[Everitt 01] Cass Everitt. "Interactive Order-Independent Transparency." Technical report, NVIDIA Corporation, 2001. Available at http://developer.nvidia.com/object/Interactive_Order_Transparency.html.

[Liu et al. 06] Bao-Quan Liu, Li-Yi Wei, and Ying-Qing Xu. "Multi-Layer Depth Peeling via Fragment Sort." Technical report, Microsoft Research Asia, 2006.

[Mammen 89] Abraham Mammen. "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique." *IEEE Computer Graphics and Applications* 9:4 (1989), 43–55.

[NVIDIA 05] NVIDIA. "GPU Programming Exposed: the Naked Truth Behind NVIDIA's Demos." Technical report, NVIDIA Corporation, 2005.



Parallelized Light Pre-Pass Rendering with the Cell Broadband Engine

Steven Tovey and Stephen McAuley

The light pre-pass renderer [Engel 08, Engel 09, Engel 09a] is becoming an ever more popular choice of rendering architecture for modern real-time applications that have extensive dynamic lighting requirements. In this chapter we introduce and describe techniques that can be used to accelerate the real-time lighting of an arbitrary three-dimensional scene on the Cell Broadband Engine without adding any additional frames of latency to the target application. The techniques described in this chapter were developed for the forthcoming PLAYSTATION3 version of *Blur* (see Figure 2.1), slated for release in 2010.[1]

2.1 Introduction

As GPUs have become more powerful, people have sought to use them for purposes other than graphics. This has opened an area of research called GPGPU (General Purpose GPU), which even major graphics card manufacturers are embracing. For example, all NVIDIA GeForce GPUs now support PhysX technology, which enables physics calculations to be performed on the GPU.

However, much less has been made of the opposite phenomenon—with the increase in speed and number of CPUs in a system, it is becoming feasible on some architectures to move certain graphics calculations from the GPU back onto the

[1] "PlayStation," "PLAYSTATION," and the "PS" family logo are registered trademarks, and "Cell Broadband Engine" is a trademark of Sony Computer Entertainment Inc. The "Blu-ray Disc" and "Blu-ray Disc" logos are trademarks. Screenshots of *Blur* appear courtesy of Activision Blizzard Inc. and Bizarre Creations Ltd.

CPU. Forthcoming hardware such as Intel's Larrabee even combines both components [Seiler 08], which will certainly lead to CPU-based approaches to previously GPU-only problems becoming more popular. Today, one such architecture is the PLAYSTATION3 where the powerful Cell Broadband Engine was designed from the outset to support the GPU in its processing activities [Shippy 09].

This paper expands upon the work of Swoboda in [Swoboda 09] and explains how the Cell Broadband Engine can be used to calculate lighting within the context of a light pre-pass rendering engine.

2.2 Light Pre-Pass Rendering

A recent problem in computer graphics has been how to construct a renderer that can handle many dynamic lights in a scene. Traditional forward rendering does not perform well with multiple lights. For example, if a pixel shader is written for up to four point lights, then only four point lights can be drawn (and no spotlights). We could either increase the number of pixel shader combinations to handle as many cases as possible, or we could render the geometry multiple times, once more for each additional light. Neither of these solutions is desirable as they increase the number of state changes and draw calls to uncontrollable levels.

A popular solution to this problem is to use a deferred renderer, which uses an idea first introduced in [Deering 88]. Instead of writing out fully lit pixels from the pixel shader, we instead write out information about the surface into a *G-Buffer*, which would include depth, normal, and material information. An example G-buffer format is shown in Figure 2.2.

Figure 2.1. A screenshot from the forthcoming *Blur*.

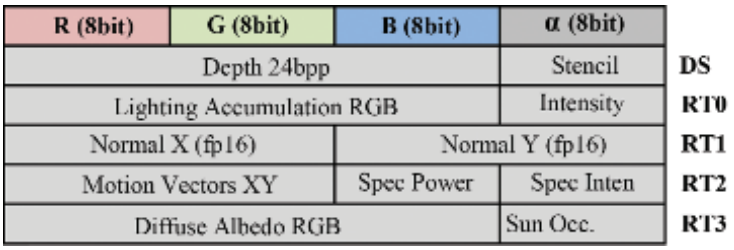


Figure 2.2. An example G-Buffer format from a deferred rendering engine (after [Valient 07]).

We then additively blend the lights into the scene, using the information provided in the G-Buffer. Thus many lights can be rendered, without additional geometry cost or shader permutations. In addition, by rendering closed volumes for each light, we can ensure that only calculations for pixels directly affected by a light are carried out. However, with deferred rendering, all materials must use the same lighting equation, and can only vary by the properties stored in the G-Buffer. There are also huge memory bandwidth costs to rendering to (and reading from) so many buffers, which increases with MSAA.

In order to solve these problems, Engel suggested the light pre-pass renderer, first online in [Engel 08] and then later published in [Engel 09], although a similar idea had been recently used in games such as *Uncharted: Drake's Fortune* [Balestra 08]. Instead of rendering out the entire G-Buffer, the light pre-pass renderer stores depth and normals in one or two render targets. The lighting phase is then performed, with the properties of all lights accumulated into a lighting buffer. The scene is then rendered for a second time, sampling the lighting buffer to determine the lighting on that pixel.

Using a *Blinn-Phong lighting model* means that the red, green, and blue channels of the lighting buffer store the diffuse calculation, while we can fit a specular term in the alpha channel, the details of which are described in [Engel 09]. This means that unlike a deferred renderer, different materials can handle the lighting values differently. This increased flexibility, combined with reduced memory bandwidth costs, has seen the light pre-pass renderer quickly increase in popularity and is now in use in many recent games on a variety of hardware platforms.

Yet the deferred renderer and light pre-pass renderer share the fact that lighting is performed in image space, and as such requires little to no rasterization. This makes the lighting pass an ideal candidate to move from the GPU back onto the CPU. Swoboda first demonstrated this method with a deferred renderer on the PLAYSTATION3 and Cell Broadband Engine in [Swoboda 09], and now we expand upon his work and apply similar techniques to the light pre-pass renderer.

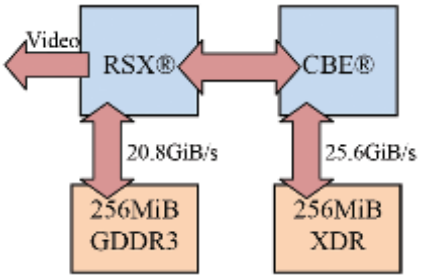


Figure 2.3. The PLAYSTATION3 architecture. (Illustration after [Möller 08, Perthuis 06]).

2.3 The PLAYSTATION3 and the CBE

Sony Computer Entertainment released the PLAYSTATION3 in 2006. It contains the Cell Broadband Engine, which was developed jointly by Sony Computer Entertainment, Toshiba Inc., and IBM Corp. [Shippy 09, Möller 08, IBM 08]. The cell is the central processing unit (CPU) of the PLAYSTATION3. In addition to the cell chip, the PLAYSTATION3 also has a GPU, the reality synthesizer (RSX). The RSX was developed by NVIDIA Corporation and is essentially a modified GeForce7800 [Möller 08]. A high-level view of the architecture can be found in Figure 2.3.

Inside the Cell chip one can find two distinctly different types of processor. There is the PowerPC Processing Element (PPE) and eight[2] pure SIMD processors [Möller 08] known as Synergistic Processing Elements (SPEs) all of which are connected by a high speed, token-ring bus known as the *element interconnect bus* (EIB; see Figure 2.4). The techniques introduced and described in this paper are chiefly concerned with the usage of the SPEs and as such further discussion of the PPE has been omitted.

One interesting quirk of the SPE is that it does not directly have access to the main address space, and instead has its own internal memory known as the *local store*. The local store on current implementations of the CBE is 256KB in size. The memory is unified, untranslatable, and unprotected [Bader 07, IBM 08] and must contain the SPE's program code, call stack, and any data that it may happen to be processing. To load or store data from or to the main address space a programmer must explicitly use the memory flow controller (MFC). Each SPE has its own MFC which is capable of queuing up to sixteen Direct Memory Accesses (DMAs) [IBM 08].

[2] One of the eight SPEs is locked out to increase chip yield and another is reserved by the Sony's Cell OS. Applications running on the PLAYSTATION3 actually have six SPEs to take advantage of.

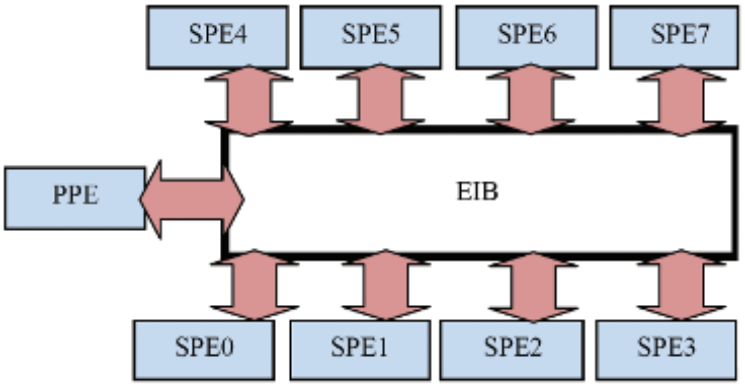


Figure 2.4. The Cell Broadband Engine (after [IBM 08]).

As the SPU ISA operates primarily on SIMD vector operands, both fixed-point and floating-point [IBM 09], it is very well equipped to process large quantities of vectorised data. It has a very large register file (4KB) which is helpful to hide the latencies of pipelined and unrolled loops, and while the local store is relatively small in capacity, it is usually sufficient to allow a programmer is able to hide the large latency of main memory accesses[3] through effective multi-buffering. Code that is to efficiently execute on the SPE should be written to play to the SPE's strengths.

A more in-depth discussion of the PLAYSTATION3 and the Cell Broadband Engine is out of the scope of this paper, interested readers can refer to IBM's website for more in depth details about the Cell chip [IBM 09], and Möller, Haines and Hoffman describe some of the PLAYSTATION3 architecture in [Möller 08].

2.4 GPU/SPE Synchronization

As the number of processors in our target platforms becomes ever greater, the need to automate the scheduling of work being carried out by these processing elements also becomes greater. This has continued to the point where game development teams now build their games and technology around the concept of the job scheduler [Capcom 06]. Our engine is no exception to this trend and the solution we propose for GPU/SPE interprocessor communication relies on close integration with such technology. It is for this reason we believe our solution to be a robust and viable solution to the problem of RSX/SPE communication that many others can easily foster into their existing scheduling frameworks.

[3] As one might expect, linear access patterns fair significantly better than random access.

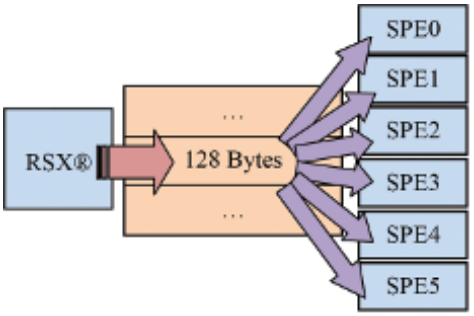


Figure 2.5. The RSX and SPE communication. The RSX writes a 128 byte value when the normal/depth buffer is available for processing. The SPEs poll the same location to know when to begin their work.

In order to perform fragment shading on the SPE without introducing unwanted latency into the rendering pipeline there needs to be a certain amount of interprocessor communication between the GPU and SPEs. This section discusses the approach we used in achieving this synchronization.

Each SPE has several memory mapped I/O (MMIO) registers it can use for interprocessor communication with other SPEs or the PPU. However, these are unfortunately not trivially writable from the RSX. An alternative approach is required in order to have the RSX signal the SPEs that the rendering of the normal/depth buffer is complete and that they can now begin their work, without having the desired SPE programs spinning on all six of the available SPEs wasting valuable processing time.

When adding a job to our job scheduler it is optionally given an address in RSX-mapped memory upon which the job is dependent. When the scheduler is pulling the next job from the job queue it polls this address to ensure that it is written to a known value by the RSX. If this is not the case, the job is skipped and the next one fetched from the queue and processed, if the location in memory is written however, then our job is free to run. This dependency is visualized in Figure 2.5.

The problem of ensuring that the GPU waits for the light buffer to be available from the SPEs is solved by a technique that is well-known to PLAYSTATION3 developers, but unfortunately we cannot disclose it here; interested developers can consult Sony's official development support website.

It is desirable for the RSX to continue doing useful work in parallel with the SPEs performing the lighting calculations. In *Blur* we are fortunate in that we have a number of additional views that are rendered which do not rely on the lighting buffer, for example, planar reflections and a rear-view mirror (in