EUROGRAPHICS Workshop on ... (200x), pp. 1–9
N.N. and N.N. (Editors)

A Digital Rights Enabled Graphics Processing System

Abstract

With the emergence of 3D graphics/arts assets trading on the Internet, to protect their intellectual property and to restrict their usage have become a new design challenge. This paper presents a novel protection model for proprietary graphics data by integrating digital rights management into the graphics processing unit and creating a digital rights enabled graphics processing system to defend piracy of entertainment software and copyrighted graphics arts. In accordance with the presented model, graphics content providers distribute scrambled or encrypted 3D graphics data along with their certified licenses. During rendering, when scrambled or encrypted graphics data, e.g. geometry or textures, are fetched by a digital rights enabled graphics processing system, it will be descrambled or decrypted. The graphics processing system also ensures that graphics data such as geometry, textures or shaders are bound only in accordance with the binding constraints designated in the licenses. Special API extensions for media/software developers are also proposed to enable our protection model. We evaluated the proposed hardware system based on cycle-based GPU simulator with configuration obtained from realistic implementation and using open source video game, Quake 3D.

1. Introduction

The industry of real time graphics applications such as video games, interactive graphical chat, 3D online games, handheld mobile gaming, etc., grows rapidly with the advancement of new graphics hardware technology. However, it remains a great technological and legal challenge to enforce digital rights protection for graphics applications. The problem becomes even more prominent with the emergence of 3D graphics/arts commerce on the Internet. In the virtual space, these trademarked and proprietary 3D models or textures in the forms of digitized sculptures [LPC*00], characters. vehicles, weapons, outfits, wallpapers, etc., possess real monetary values to online gamers, collectors and artists. According to International Intellectual Property Alliance, the loss of revenue due to piracy of entertainment software is measured in billions of dollars every year globally [IIP]. In the past, digital rights enabled silicon chips have made significant contribution to fight against piracy of music and video contents [BCK*99, HDC, SVP]. Today digital rights management (DRM) IC is widely used for protecting digital content in set-top boxes, satellite video/audio receivers, video players, mobile devices, etc. However, little research has been performed in the area of protecting digital rights of graphics data and 3D objects for the consumer market. In this paper, we explore the technologies of digital rights enabled graphics processing unit or GPU for countering piracy of real time graphics entertainment software and graphics assets.

Different from digital rights protection of media data (audio/video) that can use specially tailored hardware and protocol to enforce an end-to-end digital rights solution [HDC, SVP], enforcing the protection of graphics data with a tamper-proof silicon chip is almost impossible. In a conventional computing platform, CPU is heavily involved in pre-processing graphics data before the data are sent to the specialized graphics accelerator or GPU for accelerated rendering. Such tightly-coupled dependency and involvement of CPU in graphics processing pipeline leave many security holes to hackers for duplicating and reverse-engineering digital contents guarded by a digital rights protection system which runs the protection software on CPU.

There were techniques proposed for tracing illegal users of 3D meshes by embedding watermarks into 3D models under protection [Ben99, DGM02, PHF99]. However, watermarking does not prevent unauthorized users from duplicating, sharing or using watermarked graphics data. In general, a pure software based digital rights protection solution, as shown by history, offers very weak content protection because software components can be easily bypassed, circumvented, subverted, or reverse engineered.

The main objective of this work is to provide a tamper-proof digital rights management scheme for real time graphics data through an innovative combination of digital rights protection functionalities with GPU. The result is a *digital rights enabled graphics processing system* that provides tamper-proof processing for graphics data. Since it directly integrates digital rights protection into GPU, it leaves no loophole for hackers to launch software exploits or hardware based tampering. A hacker cannot defeat the protection offered by the presented model by simple reverse-engineering,

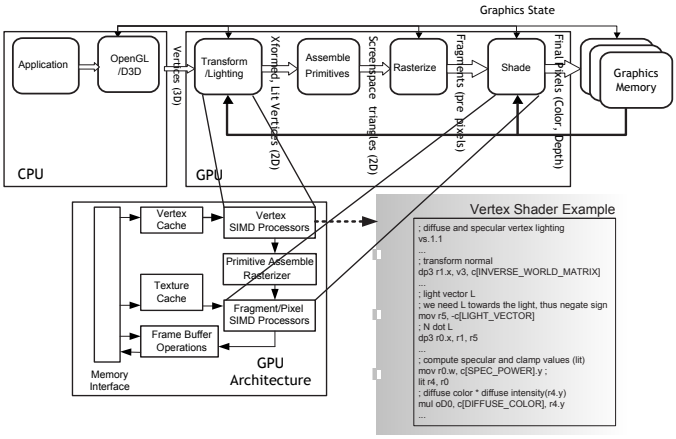

Figure 1: *3D Graphics Pipeline*

patching, altering, modifying software and data accessible by CPU.

This paper represents the first thrust to forge a common ground for digital rights community, GPU designers, and computer graphics researchers for studying digital rights issues of graphics data. The rest of the paper is organized as follows. In Section 2, we briefly discuss the background information for modern day graphics processing pipeline and GPU architecture. Section 3 presents the design of the digital rights enabled graphics processing unit. In Section 4, we discuss design challenges and graphics API extension. Finally, Section 6 concludes the paper.

2. 3D Graphics Pipeline

3D graphics processing involves projecting and rendering 3D geometry models onto a 2D display with surface detail enhancement such as texture mapping, bump mapping to create realism [FvDFH95, WDS99]. The rapid growth of 3D graphics industry to a great extent ascribes to the advancement of high performance and low cost graphics hardware and real-time graphics applications such as video games.

Figure 1 illustrates a graphics pipeline for polygon-based rendering that converts graphics input data from 3D geometry vertices into 2D image frames. It also shows the functional partitioning of CPU and GPU for state-of-the-art computing platforms. The graphics input consists of a list of geometry, denoted by triangle vertices in world coordinates. The first stage of the processing, the geometry stage, transforms each vertex from the world space into the viewer's space, culls backfaced vertices, performs perspective correction, and calculates lighting intensity for each vertex. The process is generally called geometry pipeline (transformation and lighting) and is floating-point intensive. The output from the geometry pipeline is comprised of triangles in the viewer's space. In the rasterization stage, a rasterizer determines the plane equation of each triangle surface and its corresponding coordinates on the screen. Then for each screen pixel, it computes a set of graphics parameters by interpolating vertex parameters of the triangle computed in the previous stage. The result of the rasterization stage is a set of fragments (or pixels) enclosed by each triangle. The next stage, the fragment or pixel processing stage computes the color of each pixel. When surface details enhancement is applied, each fragment takes the color values provided by texture maps into account in rendering. For each fragment, there is a set of corresponding texels. The fragment stage computes texture access addresses based on the texel coordinates and fetch the associated texture values from the textures stored in the system memory or GPU's memory. Then the fragment stage combines the fetched texture color with the interpolated lighting color to generate the fragment's final color. Finally, fragments are assembled into the frame buffer as a 2D image of color values. The graphics processing selects the final color of each pixel based on the depth (z direction) if several fragments overlap to the same location.

In early systems, CPU was responsible for completing all the graphics processing pipeline stages including geometry, rasterization, and mapping surface details. Since graphics processing exhibits substantial regularity and parallelism, CPU architects added new instruction support such as Intel's MMX to speed up the rasterization and SSE/SSE2 to enable SIMD floating-point operations for geometry. At the meanwhile, custom hardware also emerged and started to take advantages of such properties to reduce the processing overheads due to software. Today's customized GPUs, by incorporating both geometry and rasterization capability in the hardware, deliver much higher performance for graphics applications than a general purpose processor.

Figure 1 also shows the functional decoupling of 3D graphics processing between today's 3D GPU and CPU. The GPU implements each stage as a separate piece of hardware logic. Over the last few years, GPU architectures became increasingly powerful. It gradually engulfed most of the processing stages originally completed by CPU, starting from rasterization to floating-point intensive geometry processing. In addition to absorbing more graphics processing stages into silicon, the GPU architecture itself also evolved by converting some of the fixed functional stages into more flexible programmable stages. These high performance GPU architectures employ programmable pipelined SIMD engines known as vertex processor and fragment processor (or pixel processor) to process graphics data. Those engines use specially designed SIMD instructions internal to the GPU and run special *shader programs* for vertex and pixel processing. Those programs are called vertex shader or pixel shader. Figure 1 shows an example vertex shader running in vertex processor that does transformation and lighting.

With the continuing advancement of GPU hardware, the software driver on CPU became very thin layer in standard 3D pipeline. The primary task of CPU reduces to controlling of the GPU hardware, setting up, and moving graphics commands and data around, leaving extra computing power for AI, physical modeling, and gaming strategies.

3. DRM for Graphics

The industry of real time graphics entertainment applications has been mauled by wide-spread piracy since the very

beginning. Theft or piracy of graphics arts assets or even the applications themselves is relatively easy when graphics rendering relies primarily on CPU. Hackers can easily reverse-engineer a graphics application, circumvent any software based digital rights protection, recover its art assets, or make illegal copies. [KL05] discusses several potential exploits hackers can launch for violating digital rights of graphics data including attacks such as 3D model file reverse-engineering, 3D application tampering, or graphics driver tampering, etc.

The advancement of graphics hardware brings looming opportunities to enforce digital rights protection of graphics assets at silicon level because the role of CPU in a traditional graphics pipeline withers while the complexity of GPU increases over different generations. Integration of digital rights protection within GPU has the following advantages:

- **Strong digital rights protection**. Integrating digital rights protection with GPU results in a very secure content protection system. It is extremely difficult to break such a system using software based attacks or physical hardware tampering, e.g. using a logical analyzer to probe and dump signal traces at chip interconnects.
- **Sufficient protection**. Integrating digital rights protection with GPU also provides sufficient protection to graphics based entertainment software such as video games. Assume that a hacker can freely duplicate the software and violate its copyrights. Since copyrights of graphics data are enforced by the digital rights enabled GPU, the pirated software will become useless to the offenders because graphics data will not be rendered properly.
- **High performance**. Digital rights protection of graphics data requires real-time descrambling or decryption of protected data. Implementing it directly on the silicon reduces its potential negative impact on graphics processing performance.

Note that a hacker may still be able to launch sophisticated hardware attacks such as micro-probing directly on the die or other side-channel exploits e.g. differential power analysis [Wit02]. These attacks can be addressed by security countermeasures at packaging level or circuits level [Wit02]. Such studies are general to all the approaches that use silicon hardware to build a security system, thus orthogonal to the study of this work. It is also worth pointing out that different from music or video content protection, graphics protection as proposed in this paper does not significantly suffer from analog attack such as dumping display output. Reconstructing 3D models from 2D images is considered a great challenge, remaining a major research topic in computer vision. There exists no handy and reliable method for hackers to launch such sophisticated attacks to reconstruct 3D models from rendered images.

However, the nature of the research does require a cross-disciplinary collaboration in digital rights management (DRM) community, graphics researchers, and GPU architects.

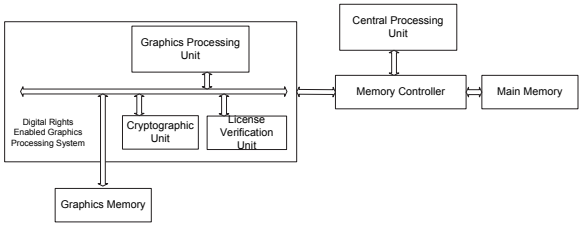

Figure 2: *Digital Rights Enabled GPU*

4. Digital Rights Aware Graphics Processing System

In this section, we outline the structure of our digital rights enabled graphics processing system. It comprises three aspects: (1) digital rights enabled GPU architecture; (2) digital rights enabled graphics API; and (3) content distribution.

4.1. Digital Rights Enabled GPU

Compared to a traditional GPU architecture, digital rights enabled GPU contains two more components, a *cryptographic unit* to decrypt or descramble protected graphics input data, and a *license verification unit* to process the licenses of graphics data. Figure 2 illustrates a digital rights enabled GPU. Similar to digital rights licenses used in other content protection cases, the graphics digital rights licenses released by their content providers specify and designate the desired usage of the graphical data. A digital rights enabled GPU features the necessary means to authenticate the licenses. During actual graphics rendering, it is guaranteed that graphics data be used strictly in accordance with the license agreement.

As discussed in Section 2, an advanced graphics processing system performs both geometric and fragment (as programmable shaders) processing in GPU. A GPU generally allows arbitrary binding of geometry data with any texture and arbitrary binding of shader programs with any geometry input. Such freedom creates potential security holes for hackers to reverse-engineer protected graphics data. For example, customized malicious shader programs can output raw, unprocessed geometry data or in a format friendly for reverse-engineering to a result buffer that a hacker can read. To reverse-engineer protected textures, a hacker can draw a proprietary textured square into the frame buffer and dump the original texture data. To prevent these exploits, bindings among geometry input, textures, and shaders must be restricted. As shown in Figure 3, such restrictions can be incorporated into the licenses of the protected graphics data.

A license of a geometry object may look like what is shown in the left-hand side of Figure 3. It comprises a name, decryption key context, signatures of scrambled geometry data, a geometry data ID, the binding constraints as to which textures or shaders can be applied to this object, and the signature of its license signed by a certified content provider. Note that the binding permission can be inherited by all the sub-classes of the object. In the example of *geometry license* for a monster 3D object in Figure 3, the geometry object is released by a certified content provider

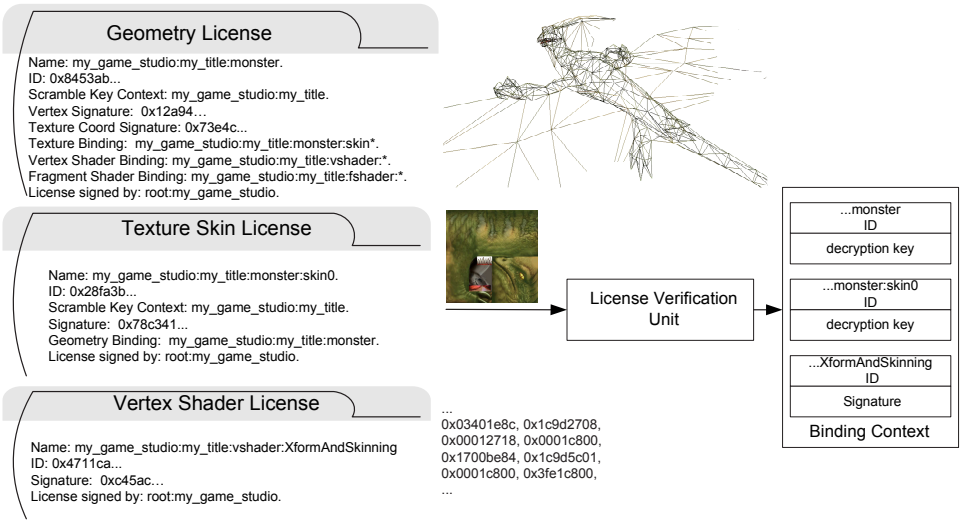

Figure 3: *Graphics Asset Licenses and Binding Context*

"my_game_studio" for a game titled "my_title". The name of the geometry object is "monster" which belongs to a root class: "my_game_studio:my_title" that includes all the graphics sub-class objects released for "my_title". According to the license, monster geometry model should be decrypted using decryption key associated with the root class "my_game_studio:my_title". The monster geometry model is allowed to bind with any texture whose name is prefixed by "my_game_studio:my_title:monster:skin". In addition, the license specifies the shader programs allowed to bind with the geometry object.

Similarly, a license for a texture map comprises similar fields. According to the license specified for monster skin texture in Figure 3, the monster skin can be decrypted using the decryption key of the root class “my_game_studio:my_title”, and is allowed to bind with the geometry object: "my_game_studio: my_title:monster". It should be understood that the binding is not a one-to-one mapping. The same texture can be bound to several graphics objects and vice versa.

Figure 3 also shows an exemplary license for vertex shader and a binding example. For a shader program, the license comprises the name of the shader, signature of its binary image, and signature of the license signed by the content provider. The described solution applies effortlessly when the shader programs are distributed in machine code binaries directly executable by the GPU. This is the case of content distribution on mobile platform. However, sometimes shader programs are distributed as source codes or in some intermediate formats that require additional compilation or translation at runtime before they are loaded to the GPU. If source code distribution is employed, it suggests that the above license binding model will be inapplicable. In this case, the content provider needs to pre-compile the shaders for different GPU targets to provide a list of shader signatures compatible with different target GPUs and device drivers for distribution along with the sources. During execution when a shader is compiled by a runtime compilation module in a host computing system, a digital rights enabled graphics processing system can verify its authenticity by computing the signature based on the given binary image and comparing it with the corresponding shader signature picked from the list of pre-computed signatures targeted for this host machine.

An alternative yet elegant solution is to integrate a secure micro-controller into the proposed system for runtime shader optimization and translation. The micro-controller resides in secure and protected domain. It authenticates shader sources before runtime optimization or translation and the object shader code is then protected with a signature generated by the micro-controller. The area cost of integrating a lean micro-controller into today’s GPU is very small provided the huge number of transistors available on modern GPUs.

Given a set of licensed geometry input, textures and shaders, the digital rights enabled graphics processing system creates a binding context that comprises decryption keys that will be used during graphics processing for decrypting protected contents such as geometry models or textures. Right side of Figure 3 illustrates this process. Furthermore, when the integrity of graphics data is a security concern, the binding context comprises signatures of the protected graphics data that can be used by the digital rights enabled graphics processing system to verify the integrity of graphics data during graphics processing. The digital rights enabled graphics processing system securely preserves confidentiality of sensitive binding context information such as decryption keys.

There is a piece of enable information included as part of a binding context to indicate whether a piece of graphics data requires decryption or authentication. When graphics

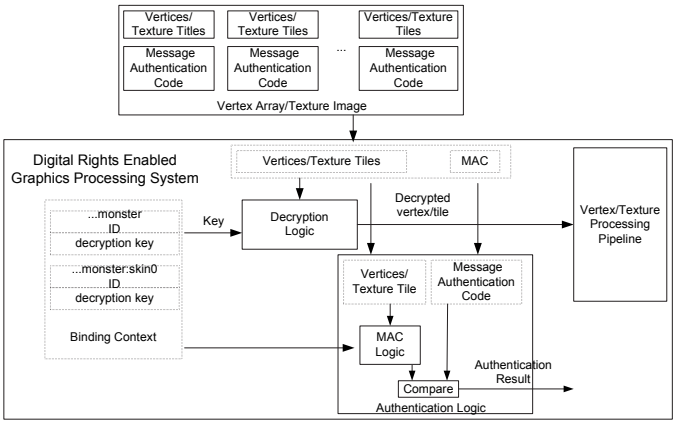

Figure 4: *Processing Scrambled Vertices*

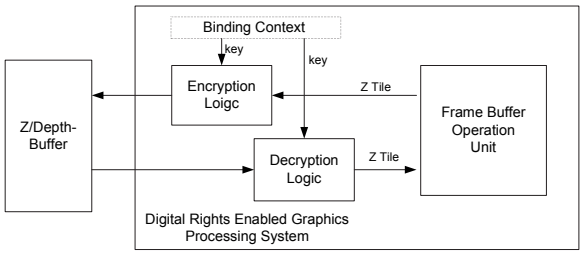

Figure 5: *Depth Buffer Protection*

data are not protected, the binding context is disabled, and all the graphics data are processed in the usual mechanism.

Figure 4 shows the components and steps for decrypting or authenticating protected geometry data during graphics processing. Each individual vertex and its attributes or a collection of vertices and their attributes are separately encrypted or scrambled. Optionally stored together with vertex attributes is a message authentication code or MAC [KBC97] that is used for optional integrity verification. During graphics processing, the digital rights enabled graphics processing system fetches encrypted vertices (or a single vertex). Based on the binding context, the cryptographic unit decrypts these vertices. Furthermore, the authentication logic computes the corresponding MAC or signature for input vertices. Then it compares the result with the MAC or signature stored together with the protected vertices. If they match, integrity of the vertices is not violated and verified.

Still, hackers could reconstruct a 3D model by dumping out the content of Z buffer that stores depth values. [KB00] presents a re-mesh technique that allows a user to reconstruct a mesh from depth buffer. Though in reality, the chance of a consumer hacker to successfully launch such an attack may be low, the issue can be addressed with protection of temporary rendering result such as depth buffer. To handle this issue in a digital rights enabled GPU, Z-buffer or other temporary result buffer can be encrypted as shown in Figure 5. The key for encrypting depth buffer or other temporary results is generated by the digital rights enabled graphics system and considered as part of a binding context.

4.2. Graphics API Extension

For a graphics application, it would be unnecessary and too restrictive to have all the graphics data protected. The content provider should make the judgment as to what are proprietary graphics data that require digital rights protection. For example, if a content provider charges a fee for advanced or customized in-game weapons and characters in virtual space, then it will make sense to have them protected. Blindly having all the graphics data protected sometimes will be an overkill on creativity because protected graphics data can only be understood and processed by the digital rights enabled graphics processing system.

Today's GPU is able to perform most of the graphics processes such as transformation, lighting, clipping, texturing, skeleton based character animation, vertex morphing, simulation, using either fixed logic or programmable shader units. Nevertheless, processes such as collision detection are still done in the CPU. Having geometry data encrypted and protected may have impact on how CPU carries out these tasks. To solve this issue, a content provider may prepare two levels of detail (LOD) for each protected geometry model using multi-resolution geometry representations [HDD*93,CVM*96,KL05], one finer LOD model with digital rights protection and the other coarser LOD models without protection. The resolution of the coarse LOD data is substantially lower than that of its finer LOD counterpart to prevent proprietary models from being disclosed via the CPU. The CPU can use the coarse LOD data to perform coarser level collision detection test and the finer LOD data can be used for rendering. Based on our study and evaluation, introducing digital rights enabled GPU should not cause any incompatibility.

glVertexAttribPointerScrambledARB()	
Argument	Attribute
index	index of the generic vertex attribute
type	data type of each component in the array (must be SCRAMBLED{234}f)
normalized	Same as glVertexAttribPointerARB
stride	Same as glVertexAttribPointerARB
pointer	Same as glVertexAttribPointerARB

Table 1: *Defines a scrambled generic vertex attribute array*

We use OpenGL to demonstrate our API extension. There will be a new geometry data type to define protected vertex data. As an example, SCRAMBLED{234}f can be used to denote encrypted vertex attributes comprising 2 to 4 floats. Note that most standard ciphers conduct basic encryption operations on a data unit of at least 64 bits or 128 bits long. If a vertex attribute does not have enough number of bits, it has to be padded or the encryption has to be carried out over a collection of vertices. Also note that this new data type is platform independent and is not affected by the endianness of the host CPU. In accordance with this new data type, a user program can declare vertex buffers of SCRAMBLED{234}f and create a vertex attribute array

of encrypted data. Figure 6 shows an example of declaring vertex buffers to store scrambled vertex data and setting up vertex array pointers using SCRAMBLED{234}f. The example uses OpenGL API with scrambled vertex format extension. It specifies monster vertex attribute arrays using **glVertexAttribPointerScrambledARB** that is an extension of **glVertexAttribPointerARB**. The new API shown in Table 1 allows a user to specify vertex array format as scrambled floats.

A typical way to store the licenses of graphics data is to store them as arrays of byte data. Similar to the case of geometry data, a new scrambled texel format, SCRAMBLED_R8G8B8A8, is defined to represent the encrypted r8g8b8a8 texture format. Note that scrambled texture format may treat a tile of texels as one unit and encrypt them as a whole. One reason of using tiles is that in many cases, the size of one texel is much smaller than the required data size of an encryption cipher. Using tile is also compatible with how GPU fetches texture data from memory into its texture cache. Figure 6 also shows how to declare a scrambled texture as SCRAMBLED_R8G8B8A8 using extended OpenGL texture API. The program calls extended **glTexImage2D**. It specifies that the monster skin's data format is SCRAMBLED_R8G8B8A8 and the data type is SCRAMBLED_BYTE, which indicates the texture's data values are scrambled bytes.

In addition, a digital rights enabled graphics driver defines the following new API calls to support binding context for graphics application developers.

API
GenBindingContext(int size, int* ptr_to_handles)
ConfigBindingContext(int handle, enum type, **int data_handle, unsigned char* license)** type = SCRAMBLED_VERTEX_ATTR0..15 type = SCRAMBLED_TEXTURE0..7 type = VERTEX_SHADER type = PIXEL_SHADER type = GEOMETRY_SHADER ... data_handle = handle to vertex buffer, texture, or shader license = license byte array
EnableBindingContext(int handle)
DisableBindingContext(int handle)
DeleteBindingContext(int handle)

Table 2: *Graphics API Extension Based on OpenGL*

The way a binding context is created, destroyed or used is similar to other OpenGL objects. As shown in Table 2, the user program can call **GenBindingContext** to create an array of binding contexts wherein size is the number of binding contexts and ptr_to_handles is a pointer to an array of binding context handles filled in by the underlying API implementation. To delete a binding context, the user program calls **DeleteBindingContext** using the binding context's handle as input. **EnableBindingContext(context_handle)** will set the bind context referenced by context_handle the current binding context. Graphics drawing commands such as commands of drawing a vertex array will consult the current binding context for digital rights information such as decryption keys for scrambled vertices and textures. **DisableBindingContext(context_handle)** will disable the binding context referenced by context_handle if it is the currently enabled context. The user program uses the **Config-**

```
// vertices and texture coordinates
unsigned char* geometry_license; // geometry license
unsigned char* scrambled_monster_vertices; // vertex data
int monster_vertex_buffer; // vertex buffer handle
unsigned char* scrambled_monster_tex_coords; // tex coords
int monster_tex_coord; // tex coord buffer handle

// texture
unsigned char* skin_license; // tex license
unsigned char* scrambled_monster_skin; // tex data
int monster_skin; // tex handle

int binding_context;

// load scrambled geometry data
glGenBuffersARB(1, &monster_vertex_buffer);
glBindBufferARB(GL_ARRAY_BUFFER_ARB, monster_vertex_buffer);
glBufferDataARB(GL_ARRAY_BUFFER_ARB,
      SIZE_OF_MONSTER_VERTICES,
      scrambled_monster_vertices, GL_STREAM_DRAW);
glVertexAttribPointerScrambledARB(0, SCRAMBLED4f,
      GL_FALSE, 0, 0); //scrambled 4f vertex coordinates
...

glGenBuffersARB(1, &monster_tex_coord);
glBindBufferARB(GL_ARRAY_BUFFER_ARB, monster_tex_coord);
glBufferDataARB(GL_ARRAY_BUFFER_ARB,
      SIZE_OF_MONSTER_TEX_COORDS,
      scrambled_monster_tex_coords, GL_STREAM_DRAW);
glVertexAttribPointerScrambledARB(8, SCRAMBLED2f,
      GL_FALSE, 0, 0); //scrambled 2f texture coordinates
...

// load scrambled texture
glGenTextures(1, &monster_skin);
glBindTexture(GL_TEXTURE_2D, monster_skin);
glTexImage2D(GL_TEXTURE_2D, 0, SCRAMBLED_R8G8B8A8,
      WIDTH,HEIGHT,
      0, SCRAMBLED_R8G8B8A8, SCRAMBLED_BYTE,
      scrambled_monster_skin);
...

// create binding contexts for monster
GenBindingContext(1, &binding_context);
ConfigBindingContext(binding_context, SCRAMBLED_VERTEX_ATTR0,
      monster_vertex_buffer, geometry_license); //vertex coordinates
ConfigBindingContext(binding_context, SCRAMBLED_VERTEX_ATTR1,
      monster_tex_coord, geometry_license); //tex coordinates
ConfigBindingContext(binding_context, SCRAMBLED_TEXTURE0,
      monster_skin, skin_license);

...

// rendering setup
glEnableVertexAttribArrayARB(0); //use monster vertex data
glEnableVertexAttribArrayARB(8); //use monster tex coords
glBindTexture(GL_TEXTURE_2D, monster_skin);
EnableBindingContext(binding_context);

// draw array command
glDrawArrays(GL_TRIANGLES, 0, VERTEX_COUNT);
```

Figure 6: *Example Program Using DRM Extended OpenGL API*

BindingContext API call to configure the current binding context.

Figure 6 illustrates how to create a binding context and configure the binding context comprising monster vertices, monster texture coordinates and monster skin. For each **ConfigBindingContext**, the user program specifies the type of graphics data, handle to the graphics data, and pointer to the data's license. **ConfigBindingContext** will trigger a sequence of processing:

(a) Setup memory references to all the required information such as the pointer to scrambled vertex buffer/texture buffer/binary shader image and the pointers to the license and the like by the software driver;

(b) Inform the digital rights enabled graphics processing system to verify the configuration and licenses;

(c) Check the integrity of the referenced graphics data by the digital rights enabled graphics processing system and authenticate the involved licenses;

(d) Verify that the binding of graphics data is consistent with the license requirement;

(e) Fill out an internal data structure comprising decryption key and IDs of the graphics data bound by the binding context.

The digital rights enabled graphics system should protect the confidentiality of the binding context. The system may store current binding contexts in an on-chip SRAM and encrypt the binding contexts when they are stored in system or video memory. Furthermore, a digital rights enabled graphics processing system considers binding context as volatile.

Global information such as content provider's certificate, certified root name and root decryption key for all graphics data of an application are considered as environment settings. As an example, user program calls **SetLicenseEnvironment(unsigned char* license)** with content provider's certificate as input. A digital rights enabled graphics processing system verifies the license's authenticity and extracts keys from the license.

4.3. Content Distribution

The presented digital rights protection technique also comprises a distribution method of protected graphics data. Distributing protected graphics data along with their licenses through a communication network is straightforward. Similar to many digital rights systems, a content distribution server can authenticate the receiver's graphics processing system and generate required licenses specifically for the targeted graphics system. The protected graphics content along with the software can also be distributed through regular retail service and stored in a regular digital storage medium such as CD-ROM or DVD. The simplest solution is to ask users to register purchased graphics application online to a registration server. The server will generate root license for the user's digital rights enabled graphics system and return it to the user.

There are several more advanced and flexible ways to distribute decryption keys and licenses of protected digital content using tamper-proof IC such as smart card. It is not the main objective of this paper to give a thorough treatment of this issue because the problem is general to all types of digital content and has been heavily studied. [AG99] describes a secure and flexible way for distributing digital rights licenses using smart card. The techniques applies to both online based and retail service based license distribution. [UKKK04] also presents a solution for distributing protected digital content and usage licenses in a more user friendly way also using smart card. Those techniques can be applied directly or with little adaptation to the presented digital right model for graphics data. The smart card based solutions also allow a user to redistribute the digital content or upgrade their GPUs. Note that it is not possible for a user to replicate smart card and the key or licenses stored inside.

5. Implementation and Performance Assessment

5.1. Implementation

The latency and throughput of decryption logic vary substantially depending on many factors such as encryption mode, cipher, authentication scheme, process technology, architecture design, etc. To best justify our performance conclusions, we use reference RTL implementations. In simulation study, we conduct sensitivity studies to capture different variations and design scenarios.

5.1.1. Cipher and Crypto Unit

The Rijndael cipher can process data blocks of 128, 192, or 256 bits by using key lengths of 128, 196 and 256 bits. It is based on a round function, which is iterated 10 times for a 128-bit length key, 12 times for a 192-bit key, and 14 times for a 256-bit key. However, Rijndael is often unrolled with each round pipelined into multiple pipeline stages (4-7) to achieve high decryption/encryption throughput [MM01, HVb, HVa]. The total area of unrolled and pipelined Rijndael is about 100K - 400K gates to achieve 15-50Gbit/sec throughput [MM01, HVb, HVa]. Based on verilog RTL implementation and synthesis results, each decryption round of pipelined AES-Rijndael takes around 2.5nsec using 0.18 μ standard cell library. The design can operate at 400MHz with area cost of around 400K gates and over 40Gbis/sec throughput. The simulated GPU comprises four independent memory partitions, each with its own crypto unit. Each crypto unit contains two AES-Rijndael blocks. This allows peak decryption or descrambling throughput of 40MB/sec. The total area cost is 3200K gates, which is negligible compared to the size of the state-of-the art GPU , which typically has transistor count of hundreds of millions.

5.1.2. Integrity Verification

Integrity verification based on MAC is often a standard operation. But variation of different MAC approaches can have significant impact on verification latency. In the reference implementation, we use standard HMAC [KBC97] for protecting integrity of graphics data blocks and shader programs. The default size of MAC is 64 bits. The reference HMAC uses standard SHA-256 algorithm [Nat]. Simulation study is based on Verilog implementation of SHA-256, synthesized using Synopsys. This design is totally asynchronous and has a gate count of 19,000 gates. The latency for

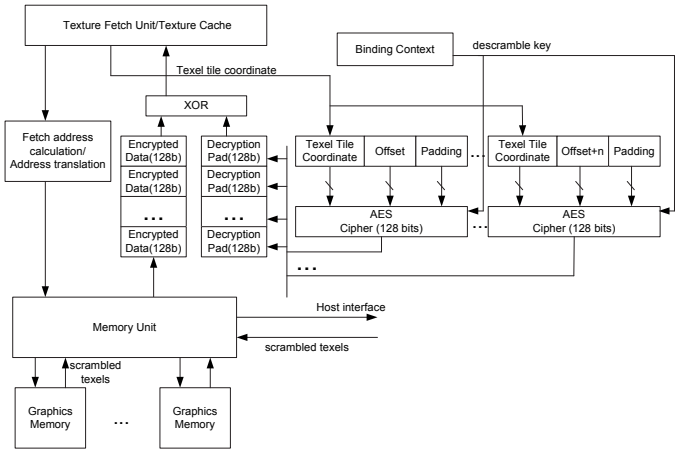

Figure 7: *Content Descrambling Block Diagram*

this design is 74ns for 512 bits of padded input (padding with the required padding in SHA-256).

5.1.3. Content Descrambling

There are multiple possible implementations of content descrambling logic depending on the design decisions of the cipher, the encryption mode, the content scrambling algorithm, etc. that will be used for graphics content scrambling. There are two obvious choices of content scrambling design. One is based on direct encryption of graphics data bits using some standard cipher such as AES. During descrambling, a decryption key will be set as part of the binding context. One main advantage of this designs is its simplicity. However, it increases the overall memory fetch latency by adding decryption latency to memory fetch delay. The other design choice is to use stream cipher-like operations for graphics data scrambling and descrambling. One of the advantages of using stream cipher-like operations is significantly reduced latency overhead as most stream ciphers allow precomputation of decryption pads or overlapping of some of decryption or descrambling processing with communication latency or data fetch latency. Figure 7 is a block diagram showing our content descrambling design. In our design, protected graphics data such as texture tiles, depth buffer tiles, or collection of geometry attributes are scrambled with pseudo-random bits via bit-wise XOR operation. Descrambling comprises simple XORing of the scrambled content with the same pseudo-random bits. The pseudo-random bits can be computed using standard ciphers such as AES by taking texture tile coordinate or depth buffer tile coordinate as input. This will generate unique pseudo-random bits for each texture tile or depth buffer tile. AES cipher takes 128-bit input and outputs a 128-bit pseudo-random bit string. When a tile of graphics data contains multiple of 128-bit blocks, inputs to the AES will comprise sub-tile coordinates or sub-tile offsets as shown in Figure 7. Furthermore, when the size of inputs is less than 128-bit, they will be padded. During decryption or descrambling, the pseudo-random bit strings or pads used for descrambling can be pre-computed in parallel with memory fetch when the texture tile coordinate or depth buffer tile is ready. This significantly reduces the descrambling latency overhead.

5.2. Simulation Environment

Our performance evaluation environment is based on Qsilver, a cycle-time modeler for state-of-the art GPU architecture [SLS04]. Qsilver comprises a front-end based on modified Chromium [HHN*02] to capture OpenGL command and data traces, and a detailed cycle based architecture simulator back-end that models the flow of data and computation through each state of a GPU pipeline, which includes vertex processing, rasterization, fragment processing, frame buffer and depth buffer updates, etc. We modified and instrumented Qsilver with simulation of our digital rights protection mechanism in order to examine the potential impact of graphics data descrambling on rendering performance. Simulation parameters are configured as close as possible to the commercial Geforce 4 GPU product line. Our Qsilver's memory model is based on GDDR3 standard. Memory parameters and latencies are set based on realistic GDDR3 performance data [Joh02].

We used open source Quake 3 Arena as evaluation workload. Protected and scrambled graphics data include static geometry data (skinned characters and mesh objects) and textures including mipmap textures. In the default protection setting, depth buffer is scrambled through encrypting of depth buffer tiles. We collected Quake traces under four different level maps. Each trace was collected after the user got into the representative part of the game.

5.3. Performance Results

First, we evaluated the impact of digital rights protection and graphics data descrambling on frame rate. Figure 8 shows normalized frame rate results under two descrambling designs. The frame rate is normalized to a baseline condition of no digital rights protection and data scrambling. The figure compares two different descrambling schemes, one based on direct encryption of graphics data blocks and the other one based on stream cipher-like encryption illustrated in Figure 7. As suggested by the figure, using stream alike descrambling incurs only small performance overhead because it hides most of the additional descrambling latency with graphics data fetch.

Descrambling latency plays an important role in the overall rendering performance of protected graphics data. Though our default simulated descrambling latency is based on realistic implementation, it is still necessary to perform sensitivity studies to show the impact of content descrambling under different descrambling latencies. Figure 9 shows normalized frame rate under the default 27.5ns descrambling latency and a hypothetical 40ns descrambling latency setting. It is clear that with the increase of descrambling latency, the rendering performance decreases.

Another factor that might affect graphics rendering performance under content protection is memory throughput requirement. With the increase of the amount of graphics data fetched by a GPU, so is the pressure on content descrambling. We studied how sensitive the rendering performance

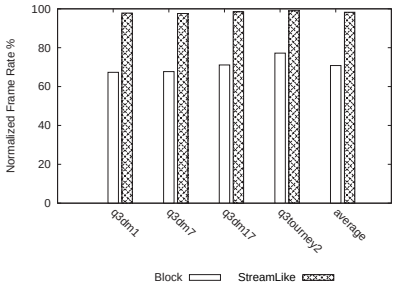

Figure 8: *Impact of Content Descrambling on Quake 3D Frame Rate Under Different Descrambling Schemes*

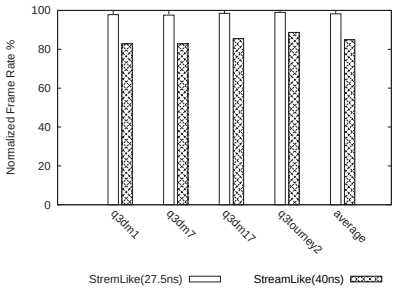

Figure 9: *Impact of Content Descrambling on Quake 3D Frame Rate Under Different AES Latencies*

is to the increase of memory fetch workload by varying the texture cache and depth buffer miss rate. Figure 10 shows the normalized frame rate results. Texture cache and depth buffer miss rate are about 10% in the medium memory pressure setting and around 20% in the high memory pressure setting. Results in Figure 10 indicate that as memory fetch throughput demand and content descrambling workload increases, rendering performance decreases.

6. Conclusion

This paper presents a hardware-based digital rights solution for protecting real time graphics assets and graphics application. It integrates digital rights functionalities with GPU to provide a strong content protection mechanism. The paper addresses hardware design issues, API extensions, and other details from the aspects of GPU design, graphics processing, and content protection and distribution.

References

[AG99] AURA T., GOLLMANN D.: Software license management with smart cards. In *Proc. USENIX Workshop on Smartcard Technology* (1999), pp. 75–85.

[BCK*99] BLOOM J. A., COX I. J., KALKER T., LINNARTZ J.-P., MILLER M. L., TRAW B.: Copy protection for dvd video. *Proc. of the IEEE, Special Issue on Identification and Protection of Multimedia Information 7*, 87 (1999), 1267–1276.

[Ben99] BENEDENS O.: Geometry-based watermarking of 3d models. *IEEE Comput. Graph. Appl. 19*, 1 (1999), 46–55.

[CVM*96] COHEN J., VARSHNEY A., MANOCHA D., TURK G., WEBER H., AGARWAL P., BROOKS F., WRIGHT W.: Simplification envelopes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM Press, pp. 119–128.

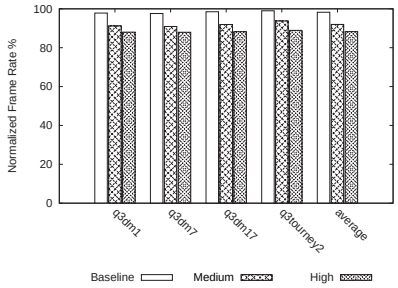

Figure 10: *Content Descrambling on Quake 3D Frame Rate*

[DGM02] DUGELAY J.-L., GARCIA E., MALLAURAN C.: Protection of 3-D object usage through texture watermarking. In *Proceedings of the 11th European Signal Processing Conference, Vol. III* (2002).

[FvDFH95] FOLEY J. D., VAN DAM A., FEINER S. K., HUGHES J. F.: *Computer Graphics: Principles and Practice in C.* Addison-Wesley Professional, 1995.

[HDC] HDCP: High-bandwidth digital content protection. *http://www.digital-cp.com/home/*.

[HDD*93] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Mesh optimization. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1993), ACM Press, pp. 19–26.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph. 21*, 3 (2002), 693–702.

[HVa] HODJAT A., VERBAUWHEDE I.: Minimum area cost for a 30 to 70 Gbits/s AES processor. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI '04), pp. 498-502*.

[HVb] HODJAT A., VERBAUWHEDE I.: Speed-area trade-off for 10 to 100 Gbits/s. In *37th Asilomar Conference on Signals, Systems, and Computer, Nov. 2003*.

[IIP] IIPA: http://www.iipa.com/.

[Joh02] JOHNSON C.: The future of memory: graphics ddr3 sdram functionality. *Designline 11*, 4 (4Q2002).

[KB00] KOBBELT L., BOTSCH M.: An interactive approach to point cloud triangulation. In *Proceedings of the Eurographics* (2000), vol. 19, pp. 479–487.

[KBC97] KRAWCZYK H., BELLARE M., CANETTI R.: HMAC: Keyed-Hashing for Message Authentication, RFC 2104, 1997.

[KL05] KOLLER D., LEVOY M.: Protecting 3d graphics content. *Communication of the ACM 48*, 6 (2005).

[LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHAD J., FULK D.: The digital michelangelo project: 3d scanning of large statues. In *Proceedings of SIGGRAPH* (2000).

[MM01] MCLOONE M., MCCANNY J. V.: High performance single-chip FPGA Rijndael algorithm implementations. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems* (2001), Springer-Verlag, pp. 65–76.

[Nat] NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY: FIPS PUB 180-2: SHA256 Hashing Algorithm.

[PHF99] PRAUN E., HOPPE H., FINKELSTEIN A.: Robust mesh watermarking. In *Proceedings of SIGGRAPH* (1999), pp. 49–56.

[SLS04] SHEAFFER J. W., LUEBKE D., SKADRON K.: A flexible simulation framework for graphics architectures. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM Press, pp. 85–94.

[SVP] SVP: Secure video processor alliance. *http://www.svpalliance.org/*.

[UKKK04] UENO M., KANBE M., KOBAYASHI T., KONDO Y.: Digital rights management technology using profile information and use authorization. *NTT Technical Review 2*, 12 (2004).

[WDS99] WOO M., DAVIS, SHERIDAN M. B.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley, 1999.

[Wit02] WITTEMAN M.: Advances in smartcard security. In *Information Security Bulletin* (July, 2002 2002).