Shader-Driven Compilation of Rendering Assets

Paul Lalonde Eric Schenk

Electronic Arts (Canada) Inc.

Abstract

Rendering performance of consumer graphics hardware benefits from pre-processing geometric data into a form targeted to the underlying API and hardware. The various elements of geometric data are then coupled with a shading program at runtime to draw the asset.

In this paper we describe a system in which pre-processing is done in a compilation process in which the geometric data are processed with knowledge of their shading programs. The data are converted into structures targeted directly to the hardware, and a code stream is assembled that describes the manipulations required to render these data structures. Our compiler is structured like a traditional code compiler, with a front end that reads the geometric data and attributes (hereafter referred to as an *art asset*) output from a 3D modeling package and shaders in a platform independent form and performs platform-independent optimizations, and a back end that performs platform-specific optimizations and generates platform-targeted data structures and code streams.

Our compiler back-end has been targeted to four platforms, three of which are radically different from one another. On all platforms the rendering performance of our compiled assets, used in real situations, is well above that of hand-coded assets.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.6 [Computer Graphics]: Methodology and Techniques—Languages; I.3.8 [Computer Graphics]: Applications—Computer Games; D.3.2 [Programming Languages]: Language Classifications—Specialized application languages

Keywords: Computer Games, Graphics Systems, Rendering, Rendering Systems

1 Introduction

The most recent generations of consumer level graphics hardware, found in such consumer devices as the Sony PlayStation2™, Microsoft XBox™, and Nintendo GameCube™, as well as in personal computers, have brought high end real-time graphics to the consumer at large. This hardware exists principally for use in video games, and this is reflected in the hardware architectures.

In consumer applications such as video games the topology of most graphical elements is fixed, unlike the case of modeling applications, such as Alias|Wavefront Maya™, SoftImage XSI™, and 3D Studio Max™. Hardware designers, both of game consoles and of graphics accelerator chipsets, have exploited this and have designed their hardware to be most efficient at rendering large constant sets of geometry than at rendering individual polygons. This is reflected in the APIs used: both Microsoft's DirectX 8 [Microsoft 2000] and OpenGL 1.1 and later [OpenGL Architecture Review Board et al. 1999] support calls for setting up arrays of input data (vertices, colours, and other per-vertex attributes, as well as index lists) that are much more efficient than single-polygon submissions. Further, groups of polygons and other rendering attributes can be collected into display lists for later atomic submission, also at much higher performance than single polygon submissions.

In a consumer application, art asset authoring is part of the development cycle. The assets are pre-processed using some set of tools into a form suitable for both the hardware and the software architecture of the application. The data pre-processes typically manipulate only the geometric elements. Setting other elements of rendering state, such as lighting, vertex and pixel shader selections, rasterization control, transformation matrices, and so forth, as well as the selection of vertex buffers and vertex layouts are handled in the runtime engine. This requires much of the knowledge about the use of the art asset to reside in code, tying the art asset closely to the programmer. Programmers often attempt to generalize this code to deal with multiple assets, at the expense of efficiency. Although shader compilers have been explored as a partial solution to this problem, no one has yet exploited knowledge of the shader to systematically optimize rendering.

Our system was driven by the following requirements:

- to render fixed topology objects efficiently with fixed shading effects;
- to support runtime modifications to the objects we draw, but not modification of their topologies;
- to exploit hardware capabilities such as vertex programs, and pixel shaders [Microsoft 2000]; and
- to have user code and art assets that are portable across hardware platforms.

We have designed and implemented a real-time rendering system comprised of a small, efficient runtime engine with a portable API, and a modular retargetable art asset compiler, called EAGL, the Electronic Arts Graphics Library. The runtime has been developed both on top of existing graphics APIs and at the driver level. An art asset is authored in some geometric modeling package, such as Alias|Wavefront Maya, 3D Studio Max, or SoftImage XSI. The asset consists of its geometry, its per-vertex attributes, and its collection of materials (surface properties, colours, texture map usages, etc.) and textures. The asset description is sufficiently rich that the object can be rendered as it should appear on final use without programmer intervention.

The EAGL architecture is based on separating the rendering primitive description from the runtime using a shader language that describes not only the shader program, but the semantics of the input data to the shader. These augmented shaders are called *render methods* and a single art asset may reference many. Conversely a render method may be used by more than one art asset.

The resulting system is fast and flexible. It has shown itself to be as fast or faster than the custom engines the various product teams were using before adopting our system. At the time of submis-

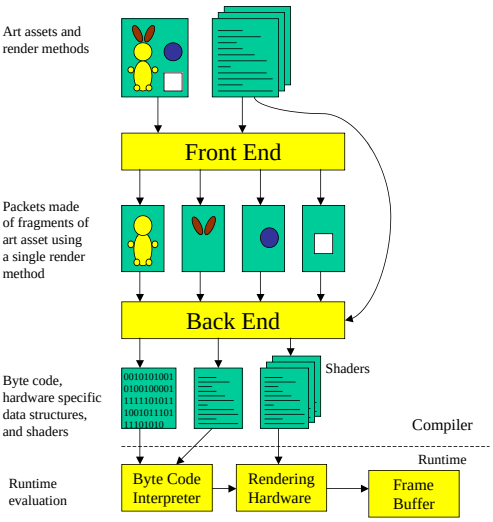

Figure 1: Art asset compilation process in context

sion the system has been used in 10 successful video game titles, in differing genres, on 4 platforms, with close to 30 more titles in development. Porting our system to a new platform involves writing new implementations of the runtime and compiler back end, and a new suite of core render methods. This amounts to approximately 3 man-months of work. In our experience this is significantly faster than porting a custom game engine and tool suite to new hardware.

1.1 Contributions

Our system, illustrated in Figure 1, makes three contributions. First, we describe an extension of shader specifications, called the *render method*, that includes declaration of shader input variables and off-line computations to generate these variables from an art asset. Second, we describe a compiler front end that takes as input polygonal geometric data and model attributes and produces segmented, geometrically optimized asset fragments, called *packets*, representing individual calls to the render methods. Alternative front ends are easily constructed to address non-polygonal data, such as spline surfaces or particle systems. Third, we describe the back end of our asset compiler that takes packets generated by the front end, and generates optimized code and data streams used to render the asset at runtime.

2 Background

Two bodies of work are relevant to the discussion of our art asset compiler. The first is the recent work done on compiling shading languages. The second relates to display lists.

2.1 Shading languages

Shading languages are an outgrowth of Cook's shade trees [Cook 1984] and Perlin's pixel stream language [Perlin 1985]. They are now most commonly used in the form of the RenderMan Shading Language [Hanrahan and Lawson 1990; Apodaca and Gritz 1999]. Shading languages have recently been adapted to real-time rendering graphics hardware applications.

Olano and Lastra [Olano and Lastra 1998] were first to describe a RenderMan-like language whose compilation is targeted to specific graphics hardware, in their case the PixelFlow system [Molnar et al. 1992]. PixelFlow is by design well suited to programmable shading but is very different from today's consumer level hardware.

id Software's Quake III product incorporates the Quake Shader Language [Jaquays and Hook 1999]. Here, shader specifications are used to control the OpenGL state machine. The shader language specifies multi-pass rendering effects involving the texture units, allowing the coupling of application variables to the parameters of the various passes.

Peercy *et al.* observed that treating the OpenGL state machine as a SIMD processor yields a framework for compiling the RenderMan Shading Language. They decompose RenderMan shaders into a series of passes of rendering, combined in the frame buffer [Peercy et al. 2000].

Recently, Proudfoot *et al.* [Proudfoot et al. 2001], have developed a shader language compiler that uses the programmable vertex shaders available in DirectX 8 [Microsoft 2000] and NVIDIA's NV_vertex_program OpenGL extension [Lindholm et al. 2001], and the the per-fragment operations provided by modern texture combiner hardware. By taking into account the multiple levels at which specifications occur (object level, vertex level, or pixel level), they successfully exploit the hardware features at those levels.

In all the above shader compilers geometric data is communicated to the shader through the underlying graphics API, as per the RenderMan model. In RenderMan both the geometry and its bindings to shaders is specified procedurally using the RenderMan Interface Specification. Likewise, Olano and Lastra's, and Proudfoot *et al.*'s systems bind shaders to geometry through the OpenGL API. This requires either an external system to manage the binding of shaders to geometry or else explicit application code per art asset to manage the bindings. These programs are more complex than they might appear at first glance, since they require both runtime code to manage the bindings, as well as synchronized tool code to generate the appropriate data for the runtime.

We have chosen to manage the binding of geometry to the shader explicitly in our art asset compiler. This provides us with three principal benefits. First, user code specific to a particular asset is only required if there are explicit exposed runtime shader parameters (§4.3). Second, as a corollary to the first, an artist can quickly iterate an art asset on the target platform without programmer intervention. Third, the runtime API is dramatically simplified, since all geometry specification and binding is performed off line. We do provide tools for generating bindings at runtime, but this use is discouraged as such *dynamic models* are considerably less efficient than our compiled models (§3.2).

2.2 Display lists

Art assets are produced in 3D modeling and animation packages. These packages are usually oriented toward interactive manipulation of geometry data and off-line rendering of the resulting objects. They have rich feature sets for manipulation of geometry, topology, shading, and animation. However, the raw output models are rarely suited to consumer level hardware. Assets must be authored with sensitivity to their eventual use in real-time consumer-level applications. The assets must be not only converted from the rich description stored by the packages, but also optimized and targeted to the hardware and software architectures of the application. These preprocessing operations range from simple data conversion through to complex re-ordering and optimization tasks. Hoppe showed how re-ordering the vertices in triangle strips could yield more efficient rendering by exploiting hardware vertex caches [Hoppe 1999]. Bogomjakov and Gotsman showed how to exploit the vertex cache using vertex meshes instead of triangle strips, without knowing

a priori the size of the cache [Bogomjakov and Gotsman 2001]. Both these approaches can yield two-fold improvements in rendering performance over using the original input data.

No matter the level of geometric optimization, however, some level of optimization of graphics hardware setup and rendering submission is required to obtain best performance. Early graphics APIs were oriented to drawing individual polygons [Barrell 1983]. The first version of OpenGL was similarly limited, leading to high function call overhead on polygon submissions. The OpenGL vertex arrays mechanism, presented in OpenGL 1.1, removed much of this overhead by allowing bulk specification of polygons [OpenGL Architecture Review Board et al. 1999]. DirectX 8's vertex streams operate on the same principle [Microsoft 2000].

Although vertex arrays speed submission of geometry data, the various state setting functions in OpenGL and DirectX 8 still require considerable overhead. Both support display lists, used to collect both geometry and state setting calls for later atomic resubmission. Although these display lists have the potential for considerable optimization at the driver level, their construction at runtime, with the ensuing performance limitations, limits the degree to which display list optimization can be taken. In particular, parameterized display lists are problematic. Although a single display list cannot be parameterized a display list may call one or more display lists which may have been re-built since the original display list, allowing simple parameterization. This architecture does not, however, allow the driver to optimize state changes across such a nested display list call, as the newly defined list may affect any of the state that had been set in the parent display list. IRIS Performer [Rohlf and Helman 1994] showed how to use a hierarchical data organization to optimize away redundant state manipulations during rendering, but forces a direct-mode geometry submission model on the application, reducing the optimizations possible in display list construction.

We have chosen to construct our rendering objects off-line, and to address parameterization of these objects through runtime linkage of external variables to our rendering objects and by exporting various part of our objects to the runtime for modification (§4.3). Both methods allow aggressive optimization around parameterized elements of our models.

3 Runtime Environment

To make our discussion of our art asset compiler concrete we briefly describe the runtime environment that our compiler targets.

3.1 The influence of the hardware environment

The target environment for our compiler consists of a particular hardware rendering platform, together with a runtime library on that platform. We implemented our runtime and compiler on four platforms: Sony PlayStation2™(PS2), Microsoft XBOX™, Nintendo GameCube™(NGC), and DirectX 8 PC platforms. Although these architectures differ substantially[Suzuoki et al. 1999; Lindholm et al. 2001; Microsoft 2000], they share some fundamental characteristics. These are:

- The CPU and GPU are separate processors that are connected by a relatively narrow bus.[1]
- The GPU is user programmable.[2]
- The platforms have varying levels of texture combining support, but all texture combining occurs as a post GPU stage with no feedback to the GPU or CPU.[3]

Keeping these characteristics in mind, we want to avoid CPU operations of the form read/compute/write/submit–to–GPU which require at least 3 times the bus traffic of a submission of static vertices to the GPU. Therefore, we privilege geometries with static vertex data and topologies. This is not to say that we do not support animation — many deformations can be applied in the GPU and pixel combiners without requiring the CPU to modify the input data. In particular, we fully support hierarchical coordinate frame animation [Stern 1983] with an arbitrary number of coordinate frames, including weighted skinning [Terzopoulos et al. 1987; Lander 1998].

3.2 The runtime API

Our runtime environment presents a small C++ API to the user, oriented toward drawing *model*s as primitive objects. A *model* is a collection of geometric primitives (e.g., triangles, b-splines, points, etc.) and state objects (rendering states, textures, etc.) bound to the shaders required to draw them with predetermined effects. A model is segmented into *geometries*, each of which can be independently turned on or off. Geometries and models map directly to art assets input to the compiler. Each run of the art asset compiler generates a single model, containing one or more geometries. Models may expose a set of control variables to the user (§4.3). Drawing a model is done by setting the required control variables and calling the model's `Draw` method. It is only necessary to set the control variables when their value changes as the system retains their values. This is the mechanism used to control animations as well as other dynamic rendering effects.

The API does not expose polygon rendering calls. The user can construct models procedurally using an API similar to the OpenGL vertex array construction [OpenGL Architecture Review Board et al. 1999]. These are referred to as *dynamic models*, and have worse rendering performance than our compiled models.

Apart from the model rendering, we also manage the usual house-keeping operations such as setting up the hardware for rendering. As well, we provide utility functions to manipulate hardware state objects and textures. Note that these objects reside in the compiled models, but may be exposed to the user. We also provide a viewport and camera abstraction as a convenience to the user, but these may be replaced if desired. Finally, we provide an animation engine that interfaces with models via their exposed control variables. The details of this system are not of interest in this paper.

4 Render Methods

A render method consists of a specification of a set of variables and a shader program that uses these variables. Render methods are designed to be sufficiently abstract that the same render method can be used over a wide range of art assets. Conversely, some render methods are written to address a problem present in a specific art asset. Our compiler architecture allows us to apply the render method to any art asset that has the data necessary to satisfy the shader input variables.

Figure 2 shows a simple example that implements gouraud shading on the PS2.[4] The `inputs` section describes the variables

[1] Even with its Unified Memory Architecture, running at AGP 4x speed, the XBOX still has limited bus bandwidth to the GPU.

[2] This is not strictly true in the case of the Nintendo GameCube, but a rich set of predefined computational elements is available.

[3] Such feedback is possible, and can be exploited by our shaders on some platforms, but are managed externally from our art asset compilation process.

[4] This is not the optimal gouraud shading code, there being many data optimizations required to achieve peak performance.

```
#include "types.rmh"
rendermethod gouraud {
  inputs {
    Coordinate4 coordinates;
    Char geometry_name;
  }
  variables {
    int nverts;
    extern volatile Matrix xform_project
      = Viewport::XFormProject;
    PackCoord3 coords[nElem]
      = PackCoordinates(coordinates);
    ColourARGB colours[nElem];
    export modifiable
           State geometry_name::state;
    noupload RenderBuffer output[nElem];
  }
  computations {
    TransformAndColour(nverts, coords,
      colours, xform_project, output);
    XGKick(geometry_name::state);
    XGKick(output);
  }
}
```

Figure 2: A simple render method

made available to the render method from the art asset. The `variables` section describes the variables used by the shader. The `computations` section describes the shader program.

4.1 Types

Because we use render methods to bind arbitrary geometric data directly to shaders, variables in both the `inputs` and `variables` sections are typed. This assures that data presented to the shader are in the form required. Types are user extensible and given in the render method specification. Type definitions are platform specific, and include such properties as the size of object in CPU memory, the size in GPU memory, hardware states required to transfer the type from CPU to GPU, and so on. The information about a type allows the compiler to manipulate the elements of the art asset without assumptions about them. We provide an identical set of base types on all platforms. A full description of the type construction system is beyond the scope of this paper.

4.2 Inputs

The `inputs` section declares the types of the data elements required by the render method. We call these *input variables*. In this example, the `coordinates` input variable is declared with type `Coordinate4`. This variable can then be referenced in a *converter*, a data manipulation program executed to construct data images for the variables section, further described in the next section. The input declaration can be accompanied by an explicit value to assign to the variable, which is used if the art asset does not provide such an input. Input data from the art asset are bound by name to each such input variable. The variables provided by our compiler include vertex and material data from the art asset, as well as arbitrary data the user may have tagged on vertices and materials. This allows easy extension of our compiler's functionality through simple naming conventions binding the user data to the render method inputs.

4.3 Variables

The `variables` section declares the data that are available to the computations. Variables are tagged with a particular type and a number of elements as an array specification. In the absence of an array specification a length of one is assumed. The special array length `nElem` is a key for a simple constraint system to maximize the number of elements within the constraints of the hardware.

All variables (save those tagged as temporaries by the `noupload` keyword) have values derived from the inputs declared in the `input` section. When an explicit *converter* is specified using the `= Converter(params,...)` syntax, the converter function (dynamically loaded from a user-extensible library) is executed with the given parameters. The parameters are themselves either the result of a conversion function, or an input variable. In the absence of an explicit converter specification, the identity converter is assumed. The converter mechanism allows simple compile-time manipulations of the data elements. Typical uses include, as in our example, data packing operations, as well as various re-indexing tasks, and various pre-processing operations such as binormal vector generation or colour balancing. Since shaders are tightly coupled to art assets through our compiler, it makes sense to move as much shader-specific pre-processing into these converters as possible, making the art conversion process as uniform as possible for differing assets and shaders.

Not all variables used by a shader can be made available through a compiler pre-process, however. For example, data such as transformation matrices are not available at compile time. They could be communicated as implicit variables. This would restrict the user from extending the set of such variables. Instead, we chose to extend our render method specification through external linkage. By adding the `extern` keyword to a variable declaration along with an assignment of the form `= variable_name`, a reference is made to an external variable named `variable_name`. In our example in Figure 2 the runtime will be responsible for replacing unresolved references to the variable named `Viewport::XFormProject` with a pointer to the actual runtime address of this variable. This external reference is resolved when the asset is loaded. We store our assets in ELF format [Tool Interface Standards 1998] and provide the external linkage through our ELF dynamic loader. Our library registers a number of variables with the dynamic loader, making transformation matrices, for example, available to render methods. The user may also, at runtime, register his own variables.

Because the GPU and CPU may operate in parallel, changes to an externally linked variable on the CPU may lead to race conditions. To eliminate the race condition without introducing undue locking restrictions, we would like copy the data elements that might produce such a race. We do not want, however, to copy all such variable data because of the overhead required. Instead we explicitly flag the elements that might induce such a race with the `volatile` keyword. The user may omit the keyword, causing use by reference rather than copy, and more efficient execution. Omitting the `volatile` keyword on externally linked variables is generally risky, there being no control over allowed modification of such variables.

Another important aspect of our variables is that they are, in general, opaque to the runtime. It is not possible to examine or set values inside our compiled art assets, since the compiler may have re-ordered, simplified, or otherwise hidden the data elements. Although this restriction leads to greater rendering efficiency, it is not sufficiently flexible. There is frequently need to examine, on the CPU rather than in the render method, data stored in a model. In many cases it is useful to export control variables, that unlike external variables, reside with and in the art asset. Then the user does not need to construct, register, and manage these variables at runtime.

Variables are declared exported in the render method by prefixing the declaration with the keyword `export`. As the compiler

emits data that is marked as exported, it adds an ELF symbol reference to the variable, along with its name, to a dictionary associated with each model. At runtime the user can query the models dictionary to find the address of a particular variable. In general this is an iteration over a number of variables with the same name. If a variable shares the same name and binary image, only one version of that variable is added to the dictionary. If the binary images differ but the names are the same, both are added, with the same name. Since these names occur in each packet compiled with the same render method, some system is required to differentiate them.

To allow the user to distinguish between these variables, a name extension mechanism is provided. String variables can be referred to in the variable name to extend the variable name at compile time. For example, in Figure 2 the `state` variable is extended with the contents of the string variable `geometry_name`. These string variables are communicated using the same mechanism as any other inputs, and so can be generated by the compiler front end, or be strings of user data attached to the material or the input model. This name extension mechanism can be used to implement a scoping system for variables. Our compiler front end provides scoping at the level of the Model, Geometry and packet. The user can easily manage additional levels of scoping by tagging their art asset in the original authoring package.

Although the exported variables dictionary returns a pointer to the exported variable, costs are associated with allowing modification of compiled data. For example, on PC class architectures, vertex data is duplicated into memory that cannot be efficiently accessed from the CPU, but is very fast for the GPU. Therefore modifications to this data by the CPU require the data to be copied again. Such performance costs led us to require a formal declaration of which elements are to be modifiable at runtime, using the `modifiable` keyword, and we prohibit modification of data not so flagged. The dictionary maintains a flag indicating the modifiable status of a variable, and enforces the restriction on modification using the C++ const mechanism. The use of the `modifiable` flag is orthogonal to the `volatile` flag. This allows use by reference of exported modifiable data when this is appropriate (infrequent changes to large data structures), while still allowing use by copy when necessary (re-use of a model in the same frame with differing parameter settings, frequent changes). This modification mechanism gives us functionality equivalent to parameterized display lists, constructed off-line in our compiler, and suitably optimized.

One simple extension made possible by the exported variables dictionary, is a runtime GUI tool to remotely view and modify the exported variables associated with a model. This allows an artist or programmer to tweak the appearance and behavior of the model without recompiling the art asset. The compiler can in turn take the saved data from this tool as input to a later compilation of the same art asset, or indeed other art assets with similar layout.

4.4 Computations

Rather than introduce a new shader language, our system uses the native shading languages on each target platform, plus a macro expansion system to connect the variables to the shader program. To make it easier to write render methods we break up complex shaders into reusable parameterized macros. The macros are defined in external files and are catenated and assembled into machine specific shader programs. A parameter passing convention binds the variables and relevant type information to the parameters of the macros. This lets the user quickly prototype new effects using existing program fragments. However, highly optimized shaders require hand crafted shader programs. Shader compilers, such as those of Proudfoot, et al. [Proudfoot et al. 2001] or Peercy, et al. [Peercy et al. 2000] could be adopted into this architecture at this level.

Our example in Figure 2 shows a simple computations section in which a transform with colour-per-vertex function is called, followed by two `XGKick` blocks. These latter PlayStation2-specific computations initiate a transfer of values from the PlayStation2 Vector Unit to Graphics Synthesiser hardware registers, kicking off the rasterization stage of the graphics pipeline.

5 Compiler Front End

The front end of the compiler takes an art asset and constructs the series of packets required to render it. Our front end deals exclusively with art assets that are composed of polygons, although other front ends have been written to deal with other kinds of data such as spline surfaces, particle systems, custom terrain meshes, and custom city rendering systems.

Our front end breaks the art asset into geometries, which have been defined by the artist. Within each geometry the polygons are collected and classified by material and vertex properties.

Materials are intrinsic to the asset editing packages we use and can contain arbitrary user defined data, as well as the standard predefined material properties. The front end is also responsible for identifying materials that are in fact identical and merging them.

A vertex consists of the usual properties, such as a position, texture coordinate set, normal, and so on, but can also include arbitrary user defined data.

Having grouped polygons into *classes* the compiler must select a render method to associate with each class. Each class consists of: a set of material properties each with a name; a collection of vertices consisting of a set of named data elements, and a collection of polygons composed from the vertices. The basic mechanism to select a render method is to iterate over a list of available render methods until one is found whose undefined inputs can be satisfied by the available data in the class. To provide finer control the material can also be tagged with a requested render method. If the requested render method's undefined inputs cannot be satisfied by the available data, then the default mechanism is applied and a warning is issued.

Once a class has been associated with a render method, the compiler must construct one or more *packets* from the data in the class. This cutting of the data into smaller packets is a reflection of hardware restrictions that limit the number data elements that can be transfered to the GPU. On some platforms this is a hard limit based on the size of GPU memory, on others it is an empirically determined point of best efficiency. Some platforms differentiate between streams of data and constant elements. For example, PlayStation2 has a fixed size limit for one packet, including matrices, state information, as well as stream oriented data such as positions, texture coordinate sets, and normals. All these elements must fit in 512 quadwords, including the output buffers. On the XBox and PC however, there is a differentiation between stream data and other values — streams may be of near-arbitrary length, but the other data, as they are stored in shader constant registers, are limited to 192 quad words per packet on XBox and 96 on PC. These restrictions are reflected in the segmentation computation. Once constructed the packets are passed to the packet compiler layer (§6).

The process of partitioning the data in the class into packets includes the process of performing triangle stripping or mesh reordering [Hoppe 1999] for efficient rendering of the underlying polygons, and may require the cloning of vertex data that must appear in more than one packet.

A significant step in our packetization process is the association of multiple coordinate frames with the vertices. Our character animation system allows for each vertex to be attached to one of a large set of coordinate frames. These coordinate frames are in turn constructed out of a linear combination of a smaller set of coordinate frames that represent an animation skeleton. Because of memory limitations on the GPU we must limit the number of unique

coordinate frames that appear in each packet. We refer to this set of coordinate frames as the *matrix set*. This transforms our mesh optimization step into a multi dimensional optimization problem: simultaneously minimize the number of vertex transforms required on the hardware, and the number of matrix sets induced. Often a strip cannot be extended with a particular triangle because this would cause the matrix set required for the packet to exceed the maximum matrix set size. Effective algorithms to solve this multi-dimensional optimization problem are an area of active research.

6 Packet Compiler

Emitting a list of packets that are interpreted at runtime leads to poor runtime performance. There are two approaches to optimizing such a scheme. The usual approach is to optimize the runtime environment, implementing such strategies as minimizing modification of hardware state, re-ordering rendering by texture usage, caching computation results for reuse and so on. However, many of these optimizations can be performed off-line because the data to be rendered is known ahead of time.

The packet compiler is responsible for transforming the packets generated by the front end into data and associated code that can be executed to render the art asset without any external intervention by a programmer. The code generated by the packet compiler is an optimized program tailored to render exactly the input art asset. Note that the packet compiler uses no information about the topology. Hence, it can be used to compile arbitrary data sets, not just polygonal data sets.

The form of the output is radically different across platforms. Despite these differences, there is a common structure in the back end of the compiler. The back end always generates a model object which contains the following: a pointer to a *byte code* stream that must be executed to render the model; a dictionary pointing to the data exported from the render methods used in the packets representing the model; and external references to imported data that will be resolved to pointers at load time. These might occur in the byte code, or in other hardware specific data structures. Additionally, the byte code contains references to the hardware specific data structures that contain the information required for rendering.

For each platform hardware specific optimizations for rendering speed are performed on the byte code and data structures generated. These optimizations largely rely on the knowledge that the rendering of a model can be treated as an atomic operation and the state of the hardware is therefore fully controlled between each packet submitted to the hardware in the rendering of the model.

The back end is structured in several passes as follows.

Pass 1: Packet ordering.
Pass 2: Variable data construction.
Pass 3: Export data accumulation.
Pass 6: Data structure generation.
Pass 5: Code generation.
Pass 6: Global optimizations of data structures and code.
Pass 7: Code and data emission.

In the remaining subsections we examine these passes in more detail. As the final pass is largely self evident we omit further description.

6.1 Packet ordering

To use the underlying hardware efficiently, we reorder packets to minimize expensive changes in the hardware state. We restrict reordering to allow the user to retain control of rendering order. In particular, we guarantee that the geometries of a model will be rendered in the order they appear in the art asset. Currently we implement a simple heuristic to minimize expensive state changes. We group packets first by geometry, then by render method, then by textures, and finally by matrix set. A more sophisticated compiler might examine the generated code stream and model the cost of its operations to determine the best possible data ordering.

6.2 Variable data construction

The front end provides the back end with a list of packets, each of which has an associated render method and set of input data. The input data is not what is ultimately fed to the shader, and therefore it must be converted into the data defined in the variables section of the render method associated with the packet. This is accomplished by executing the converter functions specified by the render method. The result is an *instantiated packet*. In an instantiated packet the data for every variable is either known, or an external symbolic reference is known that will resolve to the memory location of that data at run time. We refer to variables that have fully known data contents as *hard data*. Variables that are only defined by `extern` declarations (imported data) are called *soft data*. At this stage, the compiler also assigns symbolic names to every variables data. These symbolic names are used to refer to the memory location containing the data, and are used in the remaining passes whenever a direct reference to the data must be generated. In the case of soft data the symbolic name is the name defined by the `extern` declaration in the render method.

Although symbolic names are assigned to each block of data at this point, the data itself is neither emitted nor placed into specific data structures. This is done in the data structure generation pass, described in §6.4.

6.3 Export data accumulation

This pass accumulates the dictionary data structure associated with the model that can be used at runtime to find exported variables. The symbolic names assigned to data in the prior pass are used here to fill in pointers in the resulting dictionary.

6.4 Data structure generation

In this pass we create the rendering data structure that holds the hard and soft data refered to by the instantiated packets. On many of our target platforms, we wish to feed the underlying rendering hardware as directly as possible. This allows us to avoid device driver overhead and unnecessary manipulation of the data. We accomplish this by building the data structures in as near native form as possible.

For example, on the PS2, a chained direct memory access (DMA) unit feeds data to the GPU in parallel with CPU operations. The DMA unit supports a nested CALL structure, much like a procedure call. This allows us to pre-build large fragments of the DMA chain with the rendering data embedded in the DMA chain. One advantage of this is that the CPU need not ever touch the data in these pre-assembled DMA fragments, only chain together CALL operations to the DMA fragments at render time. Another advantage is that memory overhead required for model submission is lowered, because extra copies of the data are not required.

On the Gamecube we construct a similar data structure that feeds the hardware directly. On the XBOX and PC we pre-assemble vertex buffers and hardware command streams.

6.5 Code generation

In the code generation pass we generate a byte code program for each instantiated packet that performs the set of CPU operations required to render the data contained in the packet. In the next pass

the byte code programs are catenated and global optimizations are performed over the resulting program.

We choose to use a byte code to express these programs, rather than native assembly instructions. The overhead in interpreting the byte code is minimal and is offset by the fact that the byte code interpreter fits into the instruction cache on the CPU. In fact, on some hardware the byte code interpretation is faster than executing a stream of in-line machine instructions. This is due to a reduction in instruction cache misses and procedure calls. Furthermore, byte code has two major advantages. First, programs in the byte code are very compact. Second, because the instruction set of our byte code is very small (10 – 20 instructions, depending on the platform), it is easy to write an optimizer for the byte code.

The instruction set for a platform depends on the underlying hardware. For example, on the PS2 we submit a single DMA chain to the hardware encoding the rendering for an entire scene. The byte code instructions on this hardware perform simple operations geared toward assembling this DMA chain, as well as some more CPU intensive operations that generate data that is needed in the DMA chain. Examples of the former operations include placing a call to a fixed chunk of DMA data into the chain, and copying volatile data directly into the chain. Examples of the later include uploading a new vertex shader program to the hardware, and computing matrix set sets for an animated object from animation data. On platforms that have more of an API between us and the hardware, the byte code closely corresponds to the calls to the underlying rendering API, for example, setting of vertex streams, state setting calls, and rendering submission instructions.

6.6 Global optimizations

The specific optimizations performed in this pass are dependent upon the nature of the target platform. We classify these optimizations into two classes: data transfer optimizations and redundant code removal.

In performing data transfer optimizations we seek to remove redundant transfers of data to the GPU. This is really a special case of redundant code removal. We do this by simulating the contents of the GPU memory over the execution of the rendering of a model, and noting uploads that do not change the memory image. For example, this optimization step removes the redundant setting of the transformation matrix from one packet to the next.

Because we do not simulate the GPU execution in detail, only the uploading of data to the GPU, we require hints to tell us when the GPU will modify the contents of a memory location, forcing us to upload into that location. We have two keywords to tag a render method variable as hints to the optimizer: `noupload` and `transient`. The `noupload` keyword indicates that a variable is a temporary variable to be used by the GPU as needed. The `transient` keyword indicates a variable that must be set before the shader program is run, but that will be modified by the execution of the shader program.

Along with data upload optimization, we consider similar optimizations of machine register setting instructions and byte code instructions. For example, on the PS2 we note successive packets that occur with only a CALL instruction in their byte code and merge the DMA chains for the packets together. This can result in very large models that are submitted with only a few byte code instructions.

As another example, on the PS2 the data transfer mechanism is itself a state machine that must be appropriately set to decode the data as it is fed to the GPU. In this case we simulate the data transfer hardware to find the minimal set of register changes required to set the data transfer hardware into the desired state. This can account for as much as a 20% reduction of the rendering time on the PS2.

Many other specific optimizations are used on the various platforms we support. We omit further details due to space constraints.

7 Results and Conclusions

Table 1 summarizes some typical performance numbers achieved by our system. These figures are sustainable throughput rates for production art assets. For the sake of comparison, we also include figures for the Stanford Bunny model. The CPU is largely idle in these examples, as would be required to allow for an interactive applications use of the CPU. The bottle necks on these systems are generally in the bus to the GPU, and the transformation and rasterization engines. In some cases, better performance numbers can be achieved by intensive use of the CPU, but this would not represent the usage we expect. In our experience the performance achieved by our system is as good or better for throughput, and much better on CPU usage than the custom rendering engines it has replaced.

The system has shown itself to be easy to port to new and differing architectures. Originally designed for the PS2, ports to the XBox and GameCube took approximately 3 man months each. For each platform this involved writing a new runtime, a new compiler back end, and a suite of render methods.

Porting a product that uses our system from one platform to another has been easy. Turn around times of as little as a week have been achieved with full products. More commonly a product port takes about a month, including required revisions to art assets to take into account platform performance differences.

The render method paradigm has also proved successful, with users greatly extending the functionality of the system by writing their own render methods. This has included such features as a crowd rendering system (seen in figure 3), multiple particle systems (an example is seen in figure 4), as well as specialized multi-texture and multi-pass effects.

Additionally, when our compiler front end semantics have been insufficient to support the needs of the user, new front ends have been developed and successfully deployed. For example, the cityscape seen in figure 4 contains many objects that share substantial portions of their geometric data. A custom front end was written that allowed this data to be shared across models using the hooks provided to externally linked variables. In another case, a front end for a custom landscape patch rendering system was written and deployed in less than a week.

There are several potential future extensions to the system that we have considered. The existing mesh generation algorithm is not optimal in the presence of matrix set constraints. As previously mentioned, better algorithms for this problem are an active area of research. Our heuristic for ordering packets is simplistic at best, and should be replaced by an optimization scheme that attempt to find the ordering with minimal execution cost. One area of weakness in the current system is that dynamic models (those constructed at runtime) do not perform as efficiently as compiled models. To address this a light weight version of the compiler should be integrated into the runtime. Finally, render methods are currently platform specific. The work on shading language compilers should be adopted into our render method scheme to allow them to become cross platform specifications.

References

APODACA, A. A., AND GRITZ, L. 1999. *Advanced Renderman: Creating CGI for Motion Pictures*. Morgan Kauffman Publishers.

BARRELL, K. F. 1983. The graphical kernel system - a replacement for core. *First Australasian Conference on Computer Graphics*, 22–26.

BOGOMJAKOV, A., AND GOTSMAN, C. 2001. Universal rendering sequences for transparent vertex caching of progressive meshes. In *Proceedings of Graphics Interface 2001*, 81–90.

Platform	 Gouraud	Lit	Skinned	 Gouraud
PS2	17.0/22.6	10.9/14.7	8.5/11.5	25.2/31.8
XBox	47.2/91.4	22.4/43.4	14.2/30.3	63.9/93.8
NGC	18.7/NA	10.3/NA	7.2/NA	NA/NA
PC	24.1/46.1	15.9/20.9	5.1/10.9	26.3/36.2

Table 1: Performance figures of our system, in millions of polygons per second and vertex indices submitted per second. The player is a 3998 polygon model, drawn without texture or lighting, with textures and lighting, and skinned with textures and lighting. The bunny model is 69451 polygons. Bunny model courtesy of the Stanford Computer Graphics Laboratory. The PC is a 1.4Ghz AMD Athlon with an ATI Radeon 8500 graphics accelerator.

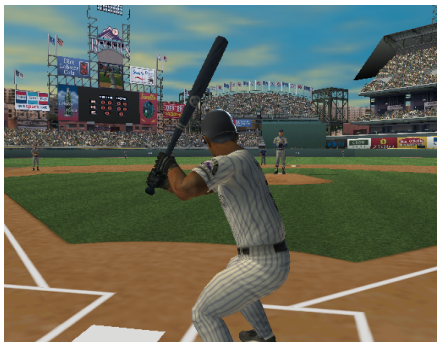

Figure 3: A screen capture of a PS2 application using our system, demonstrating skinning for the characters, a lit stadium, and a custom crowd renderer, all implemented as render methods.

Figure 4: Another scene using our system, showing cityscape generated by a user-developed alternative compiler front end, and a particle system.

COOK, R. L. 1984. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, vol. 18, 223–231.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, vol. 24, 289–298.

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. *Proceedings of SIGGRAPH 99* (August), 269–276.

JAQUAYS, P., AND HOOK, B. 1999. Q3radiant shader manual.

LANDER, J. 1998. Skin them bones: Game programming for the web generation. *Game Developer Magazine*, 11–16.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, 149–158.

MICROSOFT. 2000. *DirectX 8 Programmer's Reference*. Microsoft Press.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. Pixelflow: High-speed rendering using image composition. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, vol. 26, 231–240.

OLANO, M., AND LASTRA, A. 1998. A shading language on graphics hardware: The pixelflow shading system. In *Proceedings of SIGGRAPH 98*, ACM SIGGRAPH / Addison Wesley, Orlando, Florida, Computer Graphics Proceedings, Annual Conference Series, 159–168.

OPENGL ARCHITECTURE REVIEW BOARD, WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. 1999. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. *Proceedings of SIGGRAPH 2000* (July), 425–432.

PERLIN, K. 1985. An image synthesizer. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, vol. 19, 287–296.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, 159–170.

ROHLF, J., AND HELMAN, J. 1994. Iris performer: A high performance multiprocessing toolkit for real-time 3d graphics. In *Proceedings of SIGGRAPH 94*, ACM SIGGRAPH / ACM Press, Orlando, Florida, Computer Graphics Proceedings, Annual Conference Series, 381–395.

STERN, G. 1983. Bbop — a system for 3d keyframe figure animation. In *Introduction to Computer Animation, Course Notes 7 for SIGGRAPH 83*, 240–243.

SUZUOKI, M., KUTARAGI, K., HIROI, T., MAGOSHI, H., OKAMOTO, S., OKA, M., OHBA, A., YAMAMOTO, Y., FURUHASHI, M., TANAKA, M., YUTAKA, T., OKADA, T., NAGAMATSU, M., URAKAWA, Y., FUNYU, M., KUNIMATSU, A., GOTO, H., HASHIMOTO, K., IDE, N., MURAKAMI, H., OHTAGURO, Y., , AND AONO, A. 1999. A microprocessor with a 128-bit cpu, ten floating-point mac's, four floating-point dividers, and an mpeg-2decoder. In *IEEE Journal of Solid-State Circuits: Special issue on the 1999 ISSCC: Digital, Memory, and Signal Processing*, IEEE Solid-State Circuits Society, 1608.

TERZOPOULOS, D., PLATT, J., BARR, A., AND FLEISCHER, K. 1987. Elastically deformable models. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, vol. 21, 205–214.

TOOL INTERFACE STANDARDS. 1998. Elf: Executable and linkable format. ftp://ftp.intel.com/pub/tis.