Abstract

Recent work in shading languages and programming models for graphics hardware has given graphics processor (GPU) programmers effective abstractions for expressing computation. However, no comparable abstraction properly expresses the equally important task of data storage and access. This paper presents programmable address translation as a powerful abstraction for defining complex, point-indexable GPU data structures. This abstraction enables GPU programmers to separate algorithm and data structure definitions, greatly simplifying algorithmic development and enabling reusable and interchangeable data structures. We characterize a large body of previously published GPU data structures in terms of our abstraction and define the basic operations that these high-performance structures must support. We also present Glift, a generic template library implementation of the abstraction. We demonstrate the use of Glift in simple examples, and describe two applications not previously possible on GPUs due to the complexity of the required data structures: adaptive shadow maps and octree 3D textures. Lastly, we show that our example Glift data structures perform comparably to or better than handwritten implementations while requiring only a fraction of the programming effort.


Keywords: data structures, adaptive, multiresolution, GPGPU, adaptive shadow maps, graphics hardware

1 Introduction

The powerful new programmable features of today’s graphics hardware are rapidly having an impact on many areas of computer graphics, including interactive rendering, film rendering, simulation, and visualization. The combination of traditional graphics programming and GPU stream programming (GPGPU) is enabling, for example, interactive rendering applications to support high-quality rendering effects such as ray tracing, high dynamic range lighting, and subsurface scattering that were considered offline techniques only a few years ago.

The difficulty of programming GPUs, however, remains the primary obstacle to developers attempting to fully utilize the capabilities of GPUs. While the feature set of the microprocessor has been largely static over the past two decades, and its scalar programming model is well-known to an entire generation of programmers, the GPU’s limited, data-parallel programming model is at once both unfamiliar and complex for most of its users. The rapid advances in GPU architectures and features, while delivering both higher performance and more capabilities with each new generation, also contribute to the difficulty of efficiently programming this hardware for applications in both graphics and general-purpose computation (GPGPU) domains.

Two major research efforts have attempted to address these challenges by developing simple, powerful abstractions to hide the complexity of the underlying hardware. At a time when the programmable units of GPUs were programmed in low-level, vendor-specific assembly languages, Proudfoot et al. allowed programmers to express shading computations using a single high-level “Real-Time Shading Language” [2001]. RTSL, in turn, spurred the development of industry-standard languages like Cg [Mark et al. 2003], HLSL, and the OpenGL Shading Language. However, these languages are designed primarily for shading calculations, and assembling individual shaders to enable more complex or general-purpose computation using graphics metaphors remained a challenge. In response, Buck et al. developed an abstraction for expressing more complex and general-purpose applications while hiding the complexity of the underlying implementation [2004b]. Their Brook programming environment, in their words, “abstracts the GPU as a streaming processor.”

Together, the use of high-level programming languages and the stream programming model results in an effective abstraction for the problem of expressing computation on GPUs. However, no comparable abstraction properly expresses the equally important task of data storage and access. Today, languages like RTSL, Cg, and Brook give us powerful tools to express complex algorithms, but only a limited ability to describe the complex data structures with which these algorithms must interact.

In this paper, we present programmable address translation as an abstraction for building complex, point-indexable GPU data structures. We introduce a taxonomy for address translation and use

---

1We define an N-D point-indexable structure as a data structure that permits random access to its elements via an N-D point.
it to characterize previously published GPU data structures. We next describe Glift, our template library implementation of the abstraction, and the design strategies that enable Glift data structures to perform comparably with hand-coded versions. Two novel applications—adaptive shadow maps and octree 3D painting—demonstrate the expressive power of Glift through the use of a dynamic multiresolution adaptive GPU data structure. Finally, we present an analysis showing that our example Glift data structures perform comparably to hand-coded equivalents as well as results indicating the types of data structures that perform well on current and future graphics processors.

The contributions of this work are:

- the development of programmable address translation as a simple, powerful abstraction for GPU data structures;
- the design, implementation, and analysis of the Glift template library to implement the abstraction; and
- the demonstration of two new GPU applications that showcase the ability to use Glift to represent complex GPU data structures: interactive adaptive shadow maps and octree 3D paint.

2 Motivation

A strength of CPU programming environments is their flexible and powerful abstractions for memory access and storage. The Boost high-level CPU library, for instance, allows programmers to declare a 4D structure with the declaration `multi_array<float, 4>` without worrying about how this 4D structure is mapped onto the underlying 1D memory. With abstract structures defined by common CPU libraries like the Standard Template Library (STL) and Boost, CPU programmers can think of their problems in a virtual domain—the 2D grid, the stack, the queue—and not in the physical 1D domain imposed by the CPU’s memory and its low-level programming interfaces.

Because the GPU’s core purpose is not general-purpose computing but instead graphics, its memory systems are optimized for more specific tasks. GPU memory systems support 1D, 2D, and 3D memory primitives with memory layouts, usage requirements, performance characteristics, and special-purpose hardware functionality (like trilinear texture filtering and cubemaps) that are specific to the use of the dimensionality of the primitives. For instance, GPU programmers working in a 3D domain may wish to use the hardware features associated with a 2D memory primitive that are not available in the natural 3D memory representation, or their data may instead be best suited for a multiresolution or sparse representation in memory. Lacking the data structures that would allow them to express their computation cleanly in their virtual domains, they develop code that is often a tangled mix of computation, memory accesses in the physical domain, and memory accesses in the virtual domain.

2.1 Programmable Address Translation

Allowing GPU programmers to express their problems in their natural, virtual data domains is the key to writing high-performance, portable, flexible applications on graphics hardware. An application expressed in this way cleanly separates the problem of specifying computation from the problem of data storage and access. A well-known example is the 1D stream abstraction used by Brook [Buck et al. 2004b] and Sh [McCool et al. 2004]. The data structures must translate from the virtual memory domain exposed to the programmer to the physical memory domain available from the graphics hardware. By abstracting this mapping as a fundamental programmable primitive, programmable address translation, we can support a wide range of powerful data structures with a clean, virtual abstraction to memory.

A simple example illustrates this process. Consider a dataset defined on a regular 3D grid. The CPU programmer could simply declare a 3D array (data[x][y][z]), or construct a blocked data structure for better 3D locality. The GPU programmer has many choices, including using the native 3D texture representation in the hardware, implementing 3D access using a separate 2D texture for each slice, or mapping the entire 3D array a single, large 2D texture. All of these alternatives involve the same three-step process: describe the application’s data in its natural virtual domain; identify the most appropriate physical domain for the hardware target; and define an address translation to map the virtual addresses seen by the programmer onto the physical addresses used by the hardware. These three steps correspond exactly to the three components of our abstraction.

Expressing data structures using this abstraction has a number of benefits. With programmable address translation, we can:

- Express algorithms independently of their data’s physical data representation and implementation;
- Support spatially-optimized data structures such as sparse grids, adaptive grids, and compressed formats;
- Store an “unlimited” amount of physical data within complex structures, accessed with a single address translator; and
- “Cast” physical data to different virtual domains using multiple address translators.

We believe that just as on the CPU, GPU programmers will gain efficiency, achieve higher performance, and utilize more advanced data structures on the GPU by writing applications in their natural, virtual data domains. The goal of this work is to provide the necessary abstractions and libraries for this task.

3 Previous Work

One of the strengths of our abstraction is its ability to concisely represent a broad range of point-indexed data structures used in previous graphics and GPGPU applications. We begin by presenting our high-level classification of address translators, then describe two common address translators in detail, and finally characterize previous GPU data structure work within our classification.

3.1 Classification of Address Translators

In this section we classify the space of address translators using five key characteristics.

Representation: Analytic/Discrete Analytic address translators require no additional memory to convert virtual to physical addresses (e.g., the 1D-to-2D stream mapping defined in Brook and Sh). In contrast, discrete address translators are table-based functions requiring additional memory (e.g., a page table).

Memory Complexity: Constant/Log/Linear Memory complexity describes the amount of memory required to represent an address translator. For example, analytic mappings have memory complexity $O(1)$ and page tables are $O(n)$, where $n$ is the size of the virtual address space.

Access Complexity: Constant/Log/Linear Access complexity describes the computational cost of the address translation (e.g., tree traversals are $O(\log n)$).

Access Consistency: Uniform/Non-uniform Uniform access consistency describes address translations that always use an identical number of operations (e.g., table lookups). Examples of non-uniform translators include hash tables and linked list traversals. This characteristic is especially important for SIMD-parallel architectures.
3.2 Two Common Address Translators

As examples, we characterize two common address translators, ND-to-MD analytic mappings and page-table based mappings.

3.2.1 Analytic ND-to-MD Translators

Analytic ND-to-MD mappings are widely useful in GPU applications in which the dimensionality of the virtual domain differs from the dimensionality of the desired memory. These translators are typically implemented in one of two ways. The first form linearizes the N-D space to 1D, then distributes the the 1D space into M-D. Brook supports N-D arrays in this manner. The second approach is to directly map the N-D space into M-D memory. This is a less general approach but can result in a more efficient mapping that better preserves M-D spatial locality. For example, this approach is used in the implementation of flat 3D textures [Goodnight et al. 2003; Harris et al. 2003; Lefohn et al. 2003]. Either implementation can be implemented wholly on the GPU and is characterized as analytic and uniform with constant memory and access complexity. Lefohn et al. [2005] detail the implementation of both of these mappings for current GPUs.

3.2.2 Page Table Translators

Page-table-based mappings are single- or multi-level discrete address translators whose first level is a coarse, uniform discretization of the virtual address space (the page table). These mappings share a common set of design goals and solutions with the virtual memory systems in modern operating systems and microprocessors in that they must efficiently map a block-continuous, sparsely allocated large virtual address space onto a limited amount of physical memory [Kilburn et al. 1962; Lefohn et al. 2004]. Page-table based translators support sparse, compressed, or adaptive data structures. In addition, many page-table based structures trivially support dynamic mappings that change frequently.

Recent representative examples in computer graphics and the GPGPU community include the grid-of-lists structure [Purcell et al. 2002; Purcell et al. 2003; Johnson et al. 2004], sparse mappings [Lefohn et al. 2003; Lefohn et al. 2004], adaptive mappings [Kraus and Ertl 2002; Binotto et al. 2003; Lefebvre et al. 2004], and point-decompressible volume compressions [Schneider and Westermann 2003].

The basic address translation calculation for a 1D page table translator follows.

\[ \text{vpn} = \text{va} / \text{pageSize} \]  
\[ \text{pte} = \text{pageTable.read}(\text{vpn}) \]  
\[ \text{ppn} = \text{pte.ppn()} \]  
\[ \text{off} = \text{va} \mod \text{pageSize} \]  
\[ \text{pa} = \text{ppn} + \text{off} \]

Beginning with the above mapping, we can succinctly describe many complex structures simply as variants of this basic structure, including varying physical page sizes (grids of lists), multilevel page tables, and adaptively sized virtual or physical pages. Section 5.1 describes a new dynamic, multiresolution, adaptive GPU data structure that builds on this basic structure.

4.1 Basic Memory Operations

Writing high-performance GPU applications requires careful partitioning of the problem between the CPU and the GPU. Therefore, the required operations for GPU data structures not only must encompass efficient GPU access but also communication between the CPU and the GPU. We summarize these operations in Table 2 and note these operations are not unique to our interface but are also common in other APIs such as OpenGL/Cg and Brook.

The operations in the CPU interface are primarily concerned with moving data to and from the GPU and setting up the storage on the best way to emphasize the generality of this abstraction is to use it to characterize previous work in GPU-based data structures (Table 1). In the next section, we present a library, Glift, that implements our abstraction for generic, point-indexable data structures on graphics hardware. The data structures in Table 1, as well as many other powerful, flexible data structures, can be simply and efficiently implemented within a programmable address translation framework.

4 Glift Data Structure Model

Glift is a template library for building generic, point-indexable data structures for GPUs. The library uses the abstraction of programmable address translation to implement a wide range of data structures that support the same core of basic operations (described in Section 4.1). Examples of possible Glift structures include streams, stacks, dense and sparse arrays, compressed data, and adaptive grids. Glift is currently integrated with OpenGL and Cg with full support for both CPU and GPU versions of the data structures.

Our design goals for Glift are:

- Enabling GPU programmers to define algorithms separately from the data structures with which they interact;
- Allowing programmers to easily extend Glift to build new data structures;
- Ensuring the performance of Glift data structures is comparable to hand-written versions; and
- Providing easy integration of Glift structures into existing GPU programming environments, and make the components incrementally adoptable.

The operations in the CPU interface are primarily concerned with moving data to and from the GPU and setting up the storage on the best way to emphasize the generality of this abstraction is to use it to characterize previous work in GPU-based data structures (Table 1). In the next section, we present a library, Glift, that implements our abstraction for generic, point-indexable data structures on graphics hardware. The data structures in Table 1, as well as many other powerful, flexible data structures, can be simply and efficiently implemented within a programmable address translation framework.

4 Glift Data Structure Model

Glift is a template library for building generic, point-indexable data structures for GPUs. The library uses the abstraction of programmable address translation to implement a wide range of data structures that support the same core of basic operations (described in Section 4.1). Examples of possible Glift structures include streams, stacks, dense and sparse arrays, compressed data, and adaptive grids. Glift is currently integrated with OpenGL and Cg with full support for both CPU and GPU versions of the data structures.

Our design goals for Glift are:

- Enabling GPU programmers to define algorithms separately from the data structures with which they interact;
- Allowing programmers to easily extend Glift to build new data structures;
- Ensuring the performance of Glift data structures is comparable to hand-written versions; and
- Providing easy integration of Glift structures into existing GPU programming environments, and make the components incrementally adoptable.

4.1 Basic Memory Operations

Writing high-performance GPU applications requires careful partitioning of the problem between the CPU and the GPU. Therefore, the required operations for GPU data structures not only must encompass efficient GPU access but also communication between the CPU and the GPU. We summarize these operations in Table 2 and note these operations are not unique to our interface but are also common in other APIs such as OpenGL/Cg and Brook.

The operations in the CPU interface are primarily concerned with moving data to and from the GPU and setting up the storage on
the GPU for binding or copying. Currently, the “stream generator” operation defines the vertex data and vertex program that generates the appropriate fragments for stream-write operations; the stream operation is then run on the generated fragments. Our implementation of stream generation can change on future GPUs.

The GPU interface allows data access through three kinds of reads in addition to stream writes\(^2\). Because of the substantial bandwidth advantage of sequential (stream) reads on a modern GPU [Buck 2005; Ma and McCool 2002], data structures that are designed to use stream reads rather than random reads may enjoy higher performance than data structures that do not exhibit the same degree of locality.

We note that the basic operations we describe here are not limited to a CPU-GPU implementation. We currently support all of the same operations on a CPU-only implementation, which is useful for program development and debugging. When the program is complete, electing to use GPU memory instead is simply a matter of changing a handful of typedefs.

The basic operations in Table 2 are supported by all Glift data structures. With them, we can now specify an interface for mapping physical memory accesses into the virtual domain.

### 4.2 Glift Components

Data structures in Glift are specified by defining the three components we introduced in Section 2.1: an interface for physical memory, the specification of the address translator, and an interface for virtual memory. Figure 2 shows these components and their relationship. The source code (both CPU and GPU) for accessing Glift data structures is generated by the composition of the templated components. Each component is designed to be both composited together or used as a standalone construct. Additionally, all of the components define both a CPU and GPU implementation, thereby making it possible to easily change between CPU and GPU algorithms that use the same Glift data structures.

The following subsections describe how Glift components are assembled to create virtual memory containers and give example uses for each component. We use the simple example of a creating a 1D array virtual memory object, indexed by integers, that stores float4 values in 2D physical memory. This example can be used to define a 1D stream (the primitive data type of the stream programming model used in both GPGPU [Buck et al. 2004b] and graphics [Section 5.2] applications).

By the end of this section, we will have defined the components such that the following simple declaration and usage of our 1D array example is possible in C++:

```
VirtMemType virtMem( arraySize );
virtMem.write(0, arraySize, data);
CGParameter param = ... get Cg parameter ...
virtMem.bind( param ); // Bind data members
```

and can be used in a Cg shader as:

```
float4 main( uniform VirtMem1D array,
    float index ) : COLOR {
    return array.vTex1D(index);
}
```

### 4.2.1 Physical Memory

The PhysMem component is a lightweight abstraction around GPU texture memory. Glift supports the 1D, 2D, 3D, and cubemapped physical memory available on current GPUs as well as mipmapped versions of each. A PhysMem instance supports all basic operations defined in Section 4.1.
A PhysMem type is defined by its address type and value type. A Glift address consists of a dimensionality, value type (int or float), and an addressing mode (normalized or scaled).

The physical memory for the 1D array example defines its physical memory type as follows:

```c++
typedef PhysMemGPU<vec2i, vec4f> PhysMemType;
```

Address translators, just like physical memory objects, may also be used as stand-alone objects in shaders.

### 4.2.3 VirtMem

A VirtMem object is composed from a PhysMem object and an AddrTrans object. VirtMem objects allow virtual addressing into physical memory via the address translator.

The VirtMem type for our 1D array example simply composites the AddrTrans and PhysMem types together:

```c++
typedef VirtMemGPU<AddrTransType, PhysMemType> VirtMemType;
```

Glift's basic VirtMem type supports all basic operations except for CPU-block-read/write and creation of the stream generator. These operations are not generically defined because they depend on the continuity of the virtual-to-physical address mapping, and the interpretation of that mapping by the specific structure. Moreover, the GPU-stream-write often must include application-specific information. Support for these continuity-dependent operations is instead added in container adaptor VirtMem objects that are built atop a basic VirtMem component.

Container adaptors implement their behavior on top of an existing container. For example, in STL, stacks are container adaptors built atop either a vector or a queue. With our library, data access is often performed through a container adaptor, because there are often multiple interpretations of the same virtual memory object.

We demonstrate this idea by building a stack container adaptor on top of the 1D array defined above. Our stream-based stack replaces the array's random index accessories with pop and push operations. Pop provides a stream-read Cg interface while push is a stream-write operation. The example C++ declaration and Cg shader for pop are shown below:

```c++
StackGPU<vec4f, VirtMemType> stack{ maxSize };
```

with the corresponding usage in Cg:

```c++
void main( uniform Stack stack, 
          float outStreamIndex, 
          out float4 result : COLOR ) {
    result = stack.pop(outStreamIndex);
}
```

Glift currently supports the following container adaptors: N-D to 2D arrays, streams, stack, sparse grids implemented with a page table (see Section 3.2.2), and adaptive grids (see Section 5). Ideas for future additions include (but are not limited to) bit vectors, point-decompressible compression schemes such as vector quantization, multi-level page tables, and hash tables.

### 4.3 Shader Compilation and Parameter Binding

In order to compile Cg shaders that use Glift data structures, we must add template-like support to Cg. We approximate support for templates by overloading two Cg API calls and leveraging Cg’s interface feature [Pharr 2004]. Cg interfaces enable programmers to define abstract interface types whose definition is resolved at compile time. Shaders containing Glift data structures can be compiled either using our two new Cg API entry points or via the command-line Cg compiler with the addition of a Cg type file [NVIDIA 2003].

The new overloaded Cg API calls, cgCreateProgram and cgCreateProgramFromFile, instantiate Glift data structures by first prepending template-generated Glift Cg source code to the shader. The calls then use cgConnectParameter to specify the concrete implementation of the abstract interface declared in the shader. The CgProgram returned by these calls can be compiled, loaded, and bound like a traditional Cg program. Glift structures are bound to the shader using the syntax shown in the preceding 1D array examples.

### 4.4 Design Summary

Glift's abstraction permits the separation of shader/kernel algorithms from data structures by allowing shader/kernel designers to express algorithms in their natural address space. The shader/kernel can then be compiled with any Glift virtually-addressed data structure that has the same interface, irrespective of the physical memory representation or the specifics of the address translator.

Glift is easily extensible in two ways. First, programmable address translator kernels can be plugged into the Glift framework to create arbitrary new structures. Second, user-defined container

---

**Figure 2:** Block diagram of Glift's structure. An application can use complete data structures defined in Glift (VirtMem and container adaptor objects) or instead use Glift's lower-level AddrTrans and PhysMem objects. The Glift library is built on top of C++, Cg, and OpenGL primitives.

<table>
<thead>
<tr>
<th>Application</th>
<th>Container Adaptors</th>
<th>VirtMem</th>
<th>PhysMem</th>
<th>AddrTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eg. StreamGPU1D</td>
<td>e.g. VMGPU&lt;PhysMem, AddrTrans&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C++/Cg/OpenGL primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. glTransform()</td>
</tr>
</tbody>
</table>
adaptor classes can be application-specific while still taking advantage of all of the underlying virtual addressing infrastructure. User-defined containers may also augment the C++ and Cg interface to the structure. Although doing so limits modularity, it is often necessary and Glift does not forbid it.

We minimize performance impact through Glift’s careful integration with C++’s partial template specialization and Cg’s program specialization features, which produce efficient, entirely inline address memory access code. We show in Section 6 that our generated address translation code is as efficient as hand-written code.

Glift is currently integrated with OpenGL and Cg with full support for both CPU and GPU versions of the data structures. Integration with other languages like Sh or Brook is possible by either expressing address translators in their kernel language or converting the generated Cg source to their kernel language.

Finally, Glift is well-suited for incremental adoption. Physem objects can be used on the GPU through our abstraction or as raw texture memory. Physem and AddrTrans components can be used as stand-alone objects or as part of a Virtsem structure as appropriate. In addition, shaders that use Glift data structures can be compiled in traditional shader compilation infrastructures.

5 Case Studies

5.1 Dynamic Multiresolution Adaptive GPU Data Structures

In this section, we describe the design and use of a dynamic multiresolution adaptive data structure in two applications not previously demonstrated on GPUs: adaptive shadow maps and octree 3D paint. The data structure is defined as a Glift container adaptor, requiring only minor modifications to structures already presented thus far.

Adaptive representations of texture maps, depth maps, and simulation data are widely used in production-quality graphics and CPU-based scientific computation [Benson and Davis 2002; DeBry et al. 2002; Fernando et al. 2001; Losasso et al. 2004]. Adaptive grids make it possible for these applications to efficiently support very large resolutions by distributing grid samples based on the frequency of the stored data. Example effective resolutions include a 524,288 adaptive shadow map that consumes only 16 MB of RAM and a 10243 fluid simulation grid [Losasso et al. 2004].

Unfortunately, the complexity of adaptive-grid data structures (usually quadtrees or octrees) has, for the most part, prevented real-time graphics and GPU-based simulations from benefiting from adaptive representations.

Previous work on adaptive GPU data structures (see Table 1) includes CPU-based address translators [Carr and Hart 2004; Coome et al. 2004], static GPU-based address translators [Binotto et al. 2003; Kraus and Ertl 2002], and a GPU-based dynamic adaptive grid-of-lists [Purcell et al. 2003]. In contrast, our structure is entirely GPU-based, supports dynamic updates, and leverages the GPU’s native filtering to provide full mipmapping support (i.e., trilinear [2D] and quadtlinear [3D] filtering). Section 5.3 describes the differences between our structure and that of work done in parallel with ours by Lefebvre et al. [2004].

5.1.1 The Data Structure

We define our dynamic, multiresolution, adaptive data structure as a Glift container adaptor, built atop the 1-level page table structure defined in Section 3.2.2. As such, the structure can be described as simply a new interpretation of a page table virtual memory container (in the same way that Section 4.2.3 defines a stack on top of a 1D virtual memory definition). The structure supports adaptivity by permitting variable-sized virtual pages, while all physical pages are the same size. We support multiresolution via a mipmap hierarchy of page tables. Figure 1 shows a diagram of our structure.

Figure 3: Depiction of our node-centered adaptive grid representation. Green and red nodes indicate T-junctions (hanging nodes). The nodes on page borders are shared by the adjacent pages by over-representing the top row and right column of nodes in each page. The blue node emphasizes the complex situation at the junction of multiple levels of resolution.

The adaptive structure is declared as:

typedef AdaptiveMem< VirtMemPageTableType, PageAllocator> AdaptiveMemType;

The PageAllocator parameter defines the way in which virtual pages are mapped to physical memory. The choice of allocator determines if the structure is adaptive or uniform and whether or not it supports mipmipping (multigrid). As shown in the octree texture application, the allocator may also be application-specific. Note that the allocator is a similar construct to the allocators used in STL to generically support multiple memory models.

The adaptive address translation function differs from the one presented in Section 3.2.2 in only two small ways. First, we add a mipmap level index to the page table read:

\[ pte = \text{pageTable.read}(vpn, level) \]

Second, we support variable-sized virtual pages by storing the resolution level of virtual pages in the pte and permitting redundant page table entries (Figure 1). We thus change the offset computation to:

\[ \text{off} = \{va / 2^\text{pte.level}\} \% \text{physicalPageSize} \]

5.1.2 Adaptivity Implementation Details

Correct representation of data on an adaptive grid presents several challenges irrespective of its GPU or CPU implementation. We describe how our node-centered implementation correctly and efficiently handles T-junctions (i.e., hanging nodes), resolution changes across page boundaries, boundary conditions, and fast linear interpolation.

Adaptive grid representations generally store data at grid node positions rather than the cell-centered approach supported by OpenGL textures. A node-centered representation makes it possible to reconstruct data values at arbitrary virtual positions using a sampling scheme free of special cases, thus enabling us to correctly sample our adaptive structure using the GPU’s native linear interpolation. Cell-centered approaches must take into account a number of special cases when sampling across resolution boundaries.

While the node-centered representation makes it possible to sample the data identically at all positions, discontinuities will still occur if we do not first correct the T-junction (hanging nodes) values.
T-junctions (red and green nodes in Figure 3) arise at the boundary between resolution levels because we use the same refinement scheme on the entire grid. The data at these nodes must be the interpolated value of their neighboring coarser nodes. Before sampling our adaptive structure we enforce this constraint by again using the GPU’s native filtering to write interpolated values into the hanging nodes. This also works for higher level resolution changes between the elements (cf. Figure 3).

A node-centered representation also simplifies boundary condition support on adaptive grids. Unlike OpenGL textures, a node-centered representation contains samples on the exact boundary of the domain. As such, the position of data elements on the borders is identical and independent of the resolution (Figure 3). Dirichlet and Neumann boundary conditions are easily supported by either writing fixed values into the boundary nodes or updating the values based on the neighboring internal nodes, respectively. Our implementation supports GL_CLAMP, GL_CLAMP_TO_EDGE, and GL_REPEAT boundary modes.

The page table abstraction provides a convenient solution to the problem of reading neighboring values across resolution changes (important for high-quality filtering and scientific computation). Along each edge we must know the distance to the next regular node disregarding T-junctions, e.g. consider the regular black neighbors of the blue node in Figure 3. For a given edge, we can directly obtain the resolution level on either side of the edge from the page table. For disparate resolutions, the finer corresponds to the hanging nodes spacing while the coarser conveys the distance to the next regular node. In 2D, the overhead for this evaluation is one extra page table read, in 3D the same is true for faces and three reads are necessary at edges. This resolution information allows us to resolve hanging nodes across multi-level resolution jumps.

Lastly, in order to support native GPU linear filtering, we must share one layer of nodes between physical memory pages to ensure that samples are never read from disparate physical pages [Binotto et al. 2003; Carr and Hart 2004; Kraus and Ertl 2002]. Each time an application updates data values, the shared nodes must be updated similar to the hanging node update. Nodes that are both T-junctions and shared (green nodes in Figure 3) are correctly handled by the same interpolation scheme. The decision to represent adaptivity with variable-sized virtual pages and uniform-sized physical pages is in contrast to previous approaches. Our approach greatly simplifies support for native GPU filtering (including mipmapping) and simplifies support for dynamic structures by avoiding the bin-packing problem encountered when allocating variable-sized physical pages.

5.2 Adaptive Shadow Maps

We present the first-ever implementation of adaptive shadow maps (AMSs) that uses an entirely GPU-based ASM data structure. The ASM is represented using the multiresolution adaptive structure defined above and is shown in Figure 4.

Shadow maps, depth images rendered from the light position, offer an attractive solution to real-time shadowing because of their simplicity. Their use is plagued, however, by the problems of projective aliasing, perspective aliasing, and false self-shadowing [Fernando et al. 2001; Sen et al. 2003; Staminger and Drettakis 2002; Wimmer et al. 2004]. Adaptive shadow maps [Fernando et al. 2001] offer a rigorous solution to projective and perspective shadow map aliasing that maintains shadow mapping’s simplicity of being a purely image-based technique.

Our implementation is faithful to Fernando et al.’s original ASM algorithm [2001], but uses an entirely GPU-based ASM representation. We perform all ASM lookups and scene analysis on the GPU, and the CPU reads back only a small message (usually tens of pixels) to initiate ASM refinement and page allocations. Our implementation uses a blend of traditional graphics and GPGPU stream programming.

The ASM refinement algorithm proceeds as follows:

refineASM {  
    AnalyzeScene : Identify shadow pixels with res. mismatch  
    StreamCompaction : Pack these pixels into small stream  
    CpuReadback : Read refinement request stream  
    AllocPages : Draw new PTES into mipmap page tables  
    CreatePages : Draw depth into ASM for each new page
}

Below is an example of a Cg shader that performs an ASM lookup:

```cg
float4 main( uniform VMem2D asm,
    float3 shadowCoord ) : COLOR
{
    return asm.vTex2Ds( shadowCoord );
}
```

Note that the virtual addresses are \((s,t,z)\) shadow map coordinates and the physical addresses are 2D. The physical page size is a user-configurable parameter (typical values are 16\(^2\), 32\(^2\), or 64\(^2\)). Shadow lookups use the GPU’s native depth-compare and \(2 \times 2\) percentage-closer filtering (PCF) to return a fractional \(\text{isInLight}\) value. We improve upon Fernando et al.’s implementation by supporting trilinear (mipmapped) ASM lookups, enabling our application to transition smoothly between resolution levels with no perceptible popping.

Timing results The table below lists total frame rate including refinement (FPS) and the speed of ASM lookups relative to a standard 2048\(^2\) traditional shadow map. We list results for binearly filtered ASM (ASM L), bilinearly filtered mipmapped ASM (ASM L MN), and trilinearly filtered ASM (ASM LML).

<table>
<thead>
<tr>
<th>PageSize</th>
<th>FPS</th>
<th>ASM L</th>
<th>ASM L MN</th>
<th>ASM LML</th>
</tr>
</thead>
<tbody>
<tr>
<td>8(^2)</td>
<td>13.7</td>
<td>91%</td>
<td>77%</td>
<td>74%</td>
</tr>
<tr>
<td>16(^2)</td>
<td>15.6</td>
<td>90%</td>
<td>76%</td>
<td>73%</td>
</tr>
<tr>
<td>32(^2)</td>
<td>12.1</td>
<td>89%</td>
<td>75%</td>
<td>73%</td>
</tr>
<tr>
<td>64(^2)</td>
<td>12.9</td>
<td>89%</td>
<td>74%</td>
<td>73%</td>
</tr>
</tbody>
</table>

We achieve frame rates of 13–16 frames per second while the camera is moving for a 45k polygon model and an effective shadow...
The ASM lookup rates are bound almost entirely by the cost of the address translation instructions, thus showing that our address translator has high arithmetic intensity and is not bandwidth bound. The total framerate is dominated (~85%) by the cost of the O(logn) StreamCompaction algorithm [Hillis and Steele Jr. 1986; Horn 2005] portion of the refinement algorithm. This computation greatly reduces CPU readback cost at the expense of GPU computation (a net win), but hardware support for read-modify-write programmable blending or conditional outputs would make this operation possible with a single pass. The application becomes unnecessarily geometry bound with large models (we have tested up to one million triangles) due to the lack of frustum culling optimization used in Fernando et al.’s implementation. This is not a problem for smaller models (<100k triangles) because we minimize the number of render passes required to generate new ASM data by coalescing page requests.

5.3 Octree 3D Paint

We implement a sparse and adaptive 3D painting application that stores paint in an octree-like Glift data structure. The data structure is a 3D version of the structure described in Section 5.1.1 that supports quadrilinear (mipmap) filtering. The included movie demonstrates interactive painting of a 817K polygon model with effective resolutions varying between 643 to 20483 (also see Figure 5).

Interactive painting of surfaces with complex or unparameterized implicit surfaces is an important problem in the digital film community. Many models used in production environments are either difficult to parameterize or are unparameterized implicit surfaces. Texture atlasses offer a partial solution to the problem [Carr and Hart 2004] but cannot be easily applied to implicit surfaces. Octree textures [Benson and Davis 2002; DeBry et al. 2002] offer a more general solution by using the model’s 3D coordinates as a texture parameterization. Christensen and Batali [2004] recently refined the octree texture concept by storing pages of voxels (rather than individual voxels) at the leaves of the octree. While this texture format is now natively supported in Pixar’s Photorealistic RenderMan renderer, unfortunately, the lack of GPU support for this texture format has not been widely used. Our implementation is inspired by the octree texture techniques described by DeBry et al. [2002] and Benson and Davis [2002], including mipmap support. Our structure uses 3D virtual and physical addresses with a mipmap hierarchy of page tables and a single, small 3D physical memory buffer. 3D physical memory enables the GPU to perform native trilinear filtering. We use the normalized coordinates of the rest pose of the model.

As with the ASM structure, using the octree structure in a Cg shader is similar to a conventional 3D texture access:

```c
float4 main( uniform VMem3D octreePaint, 
float3 objCoord ) : COLOR
return octreePaint.vTex3D(objCoord);
}
```

The GPU use of our painting structure was completely defined within the core Glift functionality without application-specific modifications. In addition, we use the GPU to rasterize the model’s texture coordinates for locating brush-model intersections. The bulk of development time was spent designing proper brushing techniques and the development of appropriate mipmap-creation filters for our node-centered data representation. In this application we were able to unify the memory usage for both adaptive resolution and mipmapming levels. That is, the virtual page table allows us to share physical tiles between mip levels when the adaptive resolution is coarser than or equal to the mip level’s resolution.

The framerates for viewing textured models in our 3D paint application were determined entirely by the complexity of geometry and varied between between 15 and 80 fps with models ranging in complexity from 50K to 500K polygons. Framerates while painting depend on the size of the current brush, and we maintain highly interactive rates during painting. We evaluate the performance of our structure by comparing it to the speed of a conventional 3D texture and no texturing. We were only able to measure the performance impact of our data structure using synthetic tests similar to those shown in Figure 6. These results were invariant to page sizes between 83 and 323, and thus, lookups into our structure are bound by the address translation instructions (see Section 6).

6 Results, Discussion, and Future Work

All results in this section were computed on a 2.8 GHz Pentium 4 AXP system with 1 GB of RAM, running Microsoft Windows XP and featuring a NVIDIA GeForce 6800 with 256 MB of RAM.

Glift Performance

Programmers will not use abstractions if they impose a significant performance penalty over writing low-level code. Consequently our framework must be able to produce code with comparable efficiency to handcoded routines. We evaluate the efficiency of our code by comparing both static (instruction counts) and dynamic (runtime) metrics for three coding scenarios: a handcoded implementation of our data structure accesses and Glift code generated with Cg both before and after driver optimization.

We compare these metrics in Table 6 on three address translators: a 1D→2D stream translator, a 1-level non-adaptive page table, and the adaptive page table lookup used for adaptive shadow maps in Section 5.2 combined with an additional call to compute the page offset. This last case is a particularly difficult one for optimization because it contains a significant number of redundant instructions executed by multiple function calls.

The results for these address translators (and others that we do not show here) clearly show that the performance gap between pro-
Limitations of the Abstraction

We could express translation schemes that required either a Machine Model or MIMD, it is clear that today’s programmers using SIMD fragment processors should favor uniform computation over nonuniform computation.

Separation of Address Translation Memory from Physical Application Memory

In our implementation, address translation memory is naturally separate from physical application memory. We observe this separation has several advantages. First, we can leverage a single address translation to generate an arbitrary number of physical memory references (for instance, into different buffers). We can also “cast” physical memory from one data structure to another. The separation of the two memories also permits an efficient implementation of GPU stream-write because the physical memory can be laid out contiguously, enabling an efficient “streamify” operator. Finally, this separation may permit architectural optimizations. In CPUs, access to application memory is optimized through data caches, while access to address translation memory uses the more specialized and higher-performance translation lookaside buffer (TLB). Future GPU hardware may optimize different classes of memory accesses in a similar way.

Language Design

The high-performance computing community [Cole and Parker 2003], as well as our results for Glift, show that it is possible to express efficient data structures at a high level of abstraction by leveraging compile-time constructs such as static polymorphism (e.g., templates) and program specialization. We strongly encourage GPU language designers to add full support for static polymorphism and program specialization to their languages.

Future GPU Architectural Changes

Our implementation of Glift and our analysis of its performance allows us to make several suggestions for hardware features on future GPUs. First, hardware support of 32-bit integer data types and computation in the programmable units would both simplify address translation code and allow precise addressing of elements, avoiding the precision difficulties with floating-point addressing [Buck 2005]. Improvements in the performance of vertex textures will allow us to lift some architectural barriers to efficient implementation of GPU stream-write because the physical memory can be laid out contiguously, enabling an efficient “streamify” operator. Finally, this separation may permit architectural optimizations. In CPUs, access to application memory is optimized through data caches, while access to address translation memory uses the more specialized and higher-performance translation lookaside buffer (TLB). Future GPU hardware may optimize different classes of memory accesses in a similar way.

Figure 6: Bandwidth as a function of page size for n-level chained indirect lookups (using no address computation) and for n-level page tables using our framework. Bandwidth figures only measure the data rate of the final lookup into physical memory and not the intermediate memory references.

Table 3: Comparison of instruction counts for various compilation methods on 3 memory access routines. “Cg ops” indicates the number of operations reported by the Cg compiler before driver optimization; “HW ops” indicates the number of machine instructions (“cycles”), including multiple operations per cycle, after driver optimization, as reported by the NVIDIA shader performance tool NVShaderPerf.

<table>
<thead>
<tr>
<th>Method</th>
<th>Cg ops</th>
<th>HW ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream 1D—&gt;2D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Glift, no specialization</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Glift, with specialization</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Brook (for reference)</td>
<td>—</td>
<td>4</td>
</tr>
<tr>
<td>Handcoded Cg</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

**1D sparse, uniform 3D—>3D page table**

<table>
<thead>
<tr>
<th>Method</th>
<th>Cg ops</th>
<th>HW ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glift, no specialization</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>Glift, with specialization</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Handcoded Cg</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

**Adaptive shadow map + offset**

<table>
<thead>
<tr>
<th>Method</th>
<th>Cg ops</th>
<th>HW ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glift, no specialization</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>Glift, with specialization</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>Handcoded Cg</td>
<td>16</td>
<td>9</td>
</tr>
</tbody>
</table>

As discussed in Section 4.1, coherent memory accesses are critical to achieving high performance on a GPU memory system. This evaluation is especially critical when choosing the page size for page table address translators. We evaluate the performance of 1- and 2-level page tables built with Glift as well as n-level chained indirect lookups with no address translation between lookups (see Figure 6). We also measure reference bandwidths for both fixed-position and sequential-position direct texture accesses. These results (20 GB/sec and 9.5 GB/sec respectively) match the bandwidth tests reported by GPUBench [Buck et al. 2004a]. The page table translators are bandwidth bound when page sizes are below 16 elements and are bound by arithmetic operations beyond that. The chained indirect results show that high performance requires pages to contain at least 64 elements for a small number of directions and 256 elements for higher levels of indirection.

Machine Model

In Section 4.2.2, we noted that the AddrTrans abstraction could express translation schemes that required either a uniform or non-uniform amount of computation per element. The relative performance of uniform and non-uniform computation on a data-parallel processor such as the fragment stage in a modern GPU is highly dependent on the microarchitecture of that processor.

Today’s fragment processors have limited support for data conditions due to the fragment processor’s SIMD organization. Consequently, nonuniform accesses are inefficient because each element cannot be processed with an independent instruction stream. Instead, nonuniform computations on current GPU must be block-coherent to perform efficiently [Buck 2005]. While it is by no means clear whether the future of fragment processors is SIMD or MIMD, it is clear that today’s programmers using SIMD fragment processors should favor uniform computation over nonuniform computation.

Figure 6: Bandwidth as a function of page size for n-level chained indirect lookups (using no address computation) and for n-level page tables using our framework. Bandwidth figures only measure the data rate of the final lookup into physical memory and not the intermediate memory references.
that explicitly store connectivity between data elements such as a half-edge mesh structure. We note that current research in stream algorithms has successfully mapped many complex data structures onto the stream programming model, which may demonstrate how to refactor these complex data structures into a more GPU-friendly format. In future work, we hope to identify the generic components for data-parallel implementations of these data structures.

7 Conclusion

In this paper we have presented an abstraction for complex data structures on graphics hardware consisting of three components: physical memory, programmable address translation, and virtual memory. We then demonstrated the utility of a framework to describe and implement high-performance, complex data structures using our abstraction. Our framework, Glift, is simple to use, can describe a broad class of data structures, and can implement them efficiently. Glift enables the description and use of more complex data structures than have been previously implemented on graphics hardware, allowing the use of adaptive, multiresolution, real-time data structures in adaptive shadow maps and 3D octree paint.

We believe that the use of sophisticated and high-performance data structures on graphics hardware will only increase in importance in the coming years. In the same way that efficient implementations of data-structure libraries like STL and Boost have become integral in CPU program development, we hope that in the long run, the development of powerful GPU data structure libraries such as Glift will play a primary role in building important graphics and GPGPU algorithms on graphics hardware.

References


