Glift: Generic, Efficient, Random-Access GPU Data Structures

AARON E. LEFOHN
University of California, Davis
and
JOE KNISS
University of Utah
and
ROBERT STRZODKA
Caesar Research Institute, Bonn
and
SHUBHABRATA SENGUPTA and JOHN D. OWENS
University of California, Davis

Preprint

To Appear: *ACM Transactions on Graphics 25*(1), Jan 2006

This paper presents Glift, an abstraction and generic template library for defining complex, random-access graphics processor (GPU) data structures. Like modern CPU data structure libraries, Glift enables GPU programmers to separate algorithms from data structure definitions; thereby 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 present several new GPU data structures. The structures, a stack, quadtree, and octree, are explained using simple Glift concepts and implemented using reusable Glift components. We also describe two applications of these structures not previously demonstrated on GPUs: adaptive shadow maps and octree 3D paint. Lastly, we show that our example Glift data structures perform comparably to handwritten implementations while requiring only a fraction of the programming effort.

Categories and Subject Descriptors: I.3.1 [**Computer Graphics**]: Hardware Architecture-Graphics processors; I.3.6 [**Methodology and Techniques**]: Graphics Data Structures and Data Types; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Adaptive, adaptive shadow maps, data structures, graphics hardware, GPU, GPGPU, multiresolution, octree textures, parallel computation

1. INTRODUCTION

The computational power and programmability of today's graphics hardware is rapidly impacting many areas of computer graphics, including interactive rendering, film rendering, physical simulation, and visualization. The combination of shader-based graphics programming and general-purpose GPU programming (GPGPU) is enabling, for example, interactive rendering applications to support high-quality effects such as ray tracing, high dynamic range lighting,

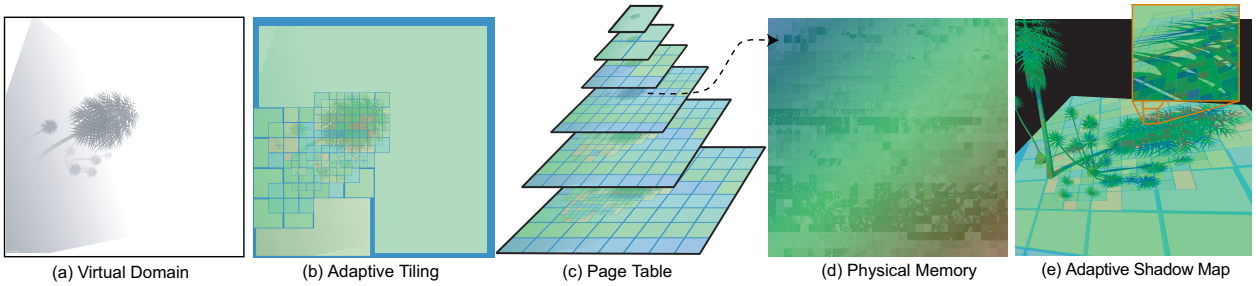


Fig. 1: Our abstraction allows GPU programmers to separate the details of data structures from the algorithms that use them. It also simplifies the description of complex structures by factoring them into orthogonal, reusable components. In this adaptive shadow map (ASM) example, the shadow data are stored in a quadtree-like structure. The programmer accesses the ASM with traditional shadow map coordinates, i.e., the structure's virtual domain (a). This domain is adaptively tiled to focus shadow resolution where required (b). The ASM performs a shadow lookup by first converting the virtual address to a physical address with an address translator (c). This address translator is a minor modification of a generic page table translator that can be used to define many other data structures. The data are then retrieved from the physical memory texture (d). The last image (e) shows an interactive rendering with the adaptive tiles colored for visualization of the shadow resolution.

and subsurface scattering that were considered offline techniques only a few years ago. In addition to rendering, GPGPU researchers are showing that the power, ubiquity and low cost of GPUs makes them an attractive alternative high-performance computing platform [Owens et al. 2005]. This literature often reports GPU speedups of more than five times over optimized CPU code. The primitive GPU programming model, however, greatly limits the ability of both graphics and GPGPU programmers to build complex applications that take full advantage of the hardware.

The GPU programming model has quickly evolved from assembly languages to high-level shading and stream processing languages [Buck et al. 2004; Kessenich et al. 2004; Mark et al. 2003; McCool et al. 2004; McCormick et al. 2004; Proudfoot et al. 2001]. Writing complex GPU algorithms and data structures, however, continues to be much more difficult than writing an equivalent CPU program. With modern CPU programming languages such as C++, programmers manage complexity by separating algorithms from data structures, using generic libraries such as the Standard Template Library (STL), and encapsulating low-level details into reusable components. In contrast, GPU programmers have very few equivalent abstraction tools available to them. As a result, GPU programs are often an inseparable tangle of algorithms and data structure accesses, are application-specific instead of generic, and rarely reuse existing code.

This paper presents a data-parallel programming abstraction that enables GPU programmers to create high-level, efficient, random-access GPU data structures. We also present *Glift*, a C++ and Cg [NVIDIA Developer Relations 2003] template library implementation of the abstraction. Glift encapsulates the GPU memory model and provides a framework for reusable, generic GPU data structure components. We demonstrate the expressive power of the abstraction by using it to characterize a large body of previous GPU data structures and by building several novel GPU data structures (stack, quadtree, and octree) and applications (adaptive shadow maps and octree 3D paint).

The main contributions of this paper are:

—The development of *programmable address translation* as a simple, powerful abstraction for GPU data structures;

—The classification of existing GPU data structures based on address translator characteristics;

—The design, implementation, and analysis of the Glift template library to implement the abstraction;

—The clarification of the GPU execution model in terms of data structure iterators; and

—The demonstration of two novel GPU applications that showcase the ability to use Glift to represent complex GPU data structures: interactive adaptive shadow maps and octree 3D paint.

1.1 Example

This section gives a brief overview of the GPU programming model and presents a small example of Glift code. GPUs use a data-parallel programming model in which a computation *pass* executes a single program (also called a *kernel* or *shader*) on all elements of a data stream in parallel. In rendering applications, the stream elements are either vertices or pixel fragments [Lindholm et al. 2001]. For general computation, the stream elements represent the data set for the particular problem [Buck et al. 2004]. Users initiate a GPU computation pass by sending data and commands via APIs such as OpenGL or DirectX, and write computational kernels in a GPU shading language (e.g., Cg, GLSL, or HLSL).

The following example Glift C++ code shows the declaration of the octree structure described in Section 6. This example allocates an octree that stores RGBA color values and has an effective maximum resolution of 2048^3. The octree is generic and reusable for various value types and is built from reusable Glift components. Note that the syntax is similar to CPU-based C++ template libraries, and the intent of the code is clear. If not implemented with the Glift template library, the intent would be obscured by numerous low-level GPU memory operations, the code would be distributed between CPU and GPU code bases, and the data structure would be hard-coded for a specific application.

```
typedef    OctreeGpu<vec3f, vec4ub> OctreeType;  // 3D float addresses, RGBA values
OctreeType octree( vec3i(2048, 2048, 2048) );    // effective size 2048^3
```

A 3D model can easily be textured by paint stored in the octree using the following Cg fragment program. Glift makes it possible for the program to be reused with any 3D structure, be it an octree, native 3D texture, or sparse volume. If implemented without Glift, this example would be hard-coded for the specific octree implementation, contain many parameters, and obscure the simple intent of reading a color from a 3D spatial data structure.

```
float4 main( uniform VMem3D paint, float3 texCoord ) : COLOR {
  return paint.vTex3D( texCoord );
}
```

By encapsulating the implementation of the entire octree structure, Glift enables portable shaders and efficient, hardware-specific implementations of data structures on current and future GPUs.

1.2 Paper Organization

This paper has four distinct target audiences: readers interested in using Glift, those interested in Glift's implementation details, CPU programmers interested in parallel data structure abstractions, and graphics researchers interested in the adaptive shadow map and octree paint applications. This section gives a brief overview of the paper organization and offers suggestions based on the reader's interests.

Section 2 describes the Glift abstractions and design goals, and Section 3 introduces the Glift programming model, illustrated by simple code examples. Section 4 delves into the implementation details of the Glift template library, including design decisions and tradeoffs. The paper demonstrates the expressiveness of the Glift abstractions in two ways. First, Section 5 provides a characterization and analysis of previous GPU data structures in terms of Glift concepts. Second, Section 6 introduces novel GPU data structures (stack, quadtree, and octree) and applications (adaptive shadow maps and octree 3D paint) built with Glift. Sections 7 and 8 present analysis of Glift's efficiency as well as insights and limitations of the abstraction.

Readers primarily interested in using Glift should start by reading Sections 2, 3 and 6. In addition, the characterization of previous structures presented in Section 5 simplifies the description of many existing structures and will interest readers looking to implement new or existing GPU data structures. Experienced GPU programmers interested in implementing features of Glift (such as templates in Cg) should read Sections 4 and 8, while parallel CPU programmers should pay particular attention to the iterator discussion in Sections 2.2.4, 6.1 and 8.3. Readers interested primarily in the adaptive shadow map and octree 3D paint applications may begin with Section 6.3, but will likely need to read Section 2 to understand the data structure details.

2. AN ABSTRACTION FOR RANDOM-ACCESS GPU DATA STRUCTURES

Glift is designed to simplify the creation and use of random-access (i.e., indexable) GPU data structures for both graphics and GPGPU programming. The abstraction combines common design patterns present in many existing GPU data structures with concepts from CPU-based data structure libraries such as the Standard Template Library (STL) and its parallel derivatives such as STAPL [An et al. 2001]. Unlike these CPU libraries, however, Glift is designed to express graphics and GPGPU data structures that are efficient on graphics hardware.

The design goals of the abstraction include:

—**Minimal abstraction of GPU memory model**
The GPU memory model spans multiple processors and languages (CPU and GPU) and is inherently data-parallel. Glift makes it possible to create complex data structures that virtualize all operations defined by the GPU memory model with a small amount of code (see Table I). Glift's abstraction level is similar to CPU data structure libraries such as the STL and Boost [2005]. It is higher-level than raw GPU memory, but low-level enough to easily integrate into existing graphics and GPGPU programming environments.

—**Separate GPU data structures and algorithms**
Separation of data structures and algorithms is a proven complexity management strategy in CPU programming. Glift's virtualization of the GPU memory model and support for generic data structure iterators bring this capability to GPU programming, thereby enabling significantly more complex GPU applications.

—**Efficiency**
GPU programming abstractions must not hinder performance. The Glift abstractions and implementation are designed to minimize abstraction penalty. The address translation and iterator abstractions generalize and encapsulate (not prevent) many existing optimization strategies.

2.1 The GPU Memory Model

Glift structures provide virtualized implementations of each operation supported by the GPU texture and vertex buffer memory model. Table I enumerates these operations and shows their syntax in OpenGL/Cg, Glift, Brook, and C. Note that the memory model contains both CPU and GPU interfaces. For example, users allocate GPU memory via a CPU-based API but read from textures in GPU-based shading code. This poses a significant challenge in creating a GPU data structure abstraction because the solution must span multiple processors and languages.

The CPU interface includes memory allocation and freeing, memory copies, binding for read or write, and mapping a GPU buffer to CPU memory. The GPU memory interface requires parameter declaration, random read access, stream read access, and stream write. Note that all copy operations are specified as parallel operations across contiguous multidimensional regions (e.g., users can copy a 16^3 data cube with one copy operation).

2.2 Glift Components

In order to separate GPU data structures into orthogonal, reusable components and simplify the explanation of complex structures, Glift factors GPU data structures into five components: virtual memory, physical memory, address translator, iterators, and container adaptors (Figure 2). This section introduces each of these components, and Section 3 presents Glift source code examples for each of them.

2.2.1 *Physical Memory.* The `PhysMem` component defines the data storage for the structure. It is a lightweight abstraction around GPU texture memory that supports the 1D, 2D, 3D, and cube-mapped physical memory available on current GPUs as well as mipmapped versions of each. A `PhysMem` instance supports all of the memory operations

Operation	OpenGL/Cg	Glift	Brook	C
CPU Interface				
Allocate	glTexImageND	class constructor	stream<>	malloc
Free	glDeleteTextures	class destructor	*implicit*	free
CPU→GPU transfer	glTexSubImageND	write	streamRead	memcpy
GPU→CPU transfer	glGetTexSubImageND	read	streamWrite	memcpy
GPU→GPU transfer	glCopyTexSubImageND	copy_from_framebuffer	*implicit*	memcpy
Bind for GPU read	glBindTextureND	bind_for_read	*implicit*	*implicit*
Bind for GPU write	glFramebufferTextureND	bind_for_write	*implicit*	*implicit*
Map to CPU	glMapBuffer	map_cpu	N/A	N/A
Map to GPU	glUnmapBuffer	unmap_cpu	N/A	N/A
GPU Interface				
Shader declaration	uniform samplerND	uniform VMemND	float<>, float[]	parameter declaration
Random read	texND(tex,coord)	vTexND	array access	array indexing
Stream read	texND(tex,streamIndex)	input iterator	stream access	read-only pointer bumping
Stream write	out floatN : COLOR	output iterator	out float<>	write-only pointer bumping

Table I: The GPU memory model described in terms of OpenGL/Cg, Glift, Brook, and C memory primitives. Note `glGetTexSubImageND` is not a true OpenGL function, but it can be emulated by attaching a texture to a framebuffer object and calling `glReadPixels`.

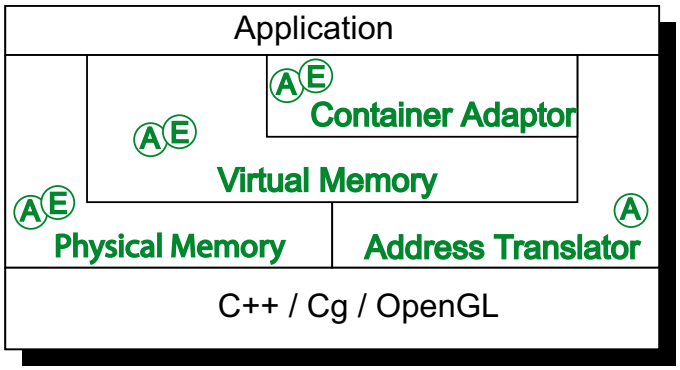


Fig. 2: Block diagram of Glift components (shown in green). Glift factors GPU data structures into a virtual (problem) domain, physical (data) domain, and an address translator that maps between them. Container adaptors are high-level structures that implement their behavior atop an existing structure. Glift structures support CPU and GPU iteration with address iterators (circled A) and element iterators (circled E). Applications can use high-level, complete Glift data structures or use Glift's lower-level `AddrTrans` and `PhysMem` components separately. The Glift library is built on top of C++, Cg, and OpenGL primitives.

defined in Table I. Users choose the type of `PhysMem` that lets them most efficiently exploit the hardware features required by a data structure. This choice is often made irrespective of the dimensionality of the virtual domain. For example, if a 3D algorithm requires efficient `bind_for_write`, current hardware mandates that the structure use 2D physical memory. If, however, fast trilinear filtering is more important than fast write operations, the user selects 3D physical memory.

2.2.2 *Virtual Memory.* The `VirtMem` component defines the programmer's interface to the data structure and is selected based on the algorithm (problem) domain. For example, if an algorithm requires a 3D data structure, the `VirtMem` component will be 3D, irrespective of the `PhysMem` type. In Glift, a generic `VirtMem` component combines a physical memory and address translator to create a virtualized structure that supports all of the operations of the GPU memory model listed in Table I. An important consequence of this feature is that `VirtMem` and `PhysMem`

components are interchangeable, making it possible for users to build complex structures by composing `VirtMem` types.

2.2.3 *Address Translator.* A Glift address translator is a mapping between the physical and virtual domains. While conceptually simple, address translators are the core of Glift data structures and define the small amount of code required to virtualize all of GPU memory operations. Address translators support mapping of single points as well as contiguous ranges. Point translation enables random-access reads and writes, and range translation allows Glift to support efficient block operations such as copy, write, and iteration. Example address translators include the ND-to-2D translator used by Brook to represent N-D arrays, page-table based translators used in sparse structures, and tree-based translators recently presented in the GPU ray tracing literature.

Just as users of the STL choose an appropriate container based on the required interface and performance considerations, Glift users select a translator based on a set of features and performance criteria. Section 5 presents a taxonomy of previous GPU data structures in terms of Glift components and these characteristics. The following six characteristics of address translators were selected based on patterns in previous GPU data structures and the performance considerations for current graphics hardware:

—**Memory Complexity: Constant/Log/Linear**
How much memory is required to represent the address translator function? Fully procedural mappings, for example, have $O(1)$ memory complexity and page tables are $O(n)$, where n is the size of the virtual address space.

—**Access Complexity: Constant/Log/Linear**
What is the average cost of performing an address translation?

—**Access Consistency: Uniform/Non-uniform**
Does address translation always require the exact same number of instructions? This characteristic is especially important for SIMD-parallel architectures. Non-uniform translators, such as hash tables or trees, require a varying number of operations per translation.

—**Location: CPU/GPU**
Is the address translator on the CPU or GPU? CPU translators are used to represent structures too complex for the GPU or as an optimization to pre-compute physical memory addresses.

—**Mapping : Total/Partial and One-to-one/Many-to-one**
Is all of the virtual domain mapped to the physical domain? Sparse data structures use partial-mappings. Does each point in the virtual domain map to a single unique point in the physical domain? Adaptive mappings optimize memory usage by mapping coarse regions of the virtual domain to less physical memory.

—**Invertible**
Does the address translator support both virtual-to-physical and physical-to-virtual mappings? The latter is required for GPU iteration over the structure (i.e., GPGPU computation).

2.2.4 *Iterators.* The `PhysMem`, `VirtMem` and `AddrTrans` components are sufficient to build texture-like read-only data structures but lack the ability to specify computation over a structure's elements. Glift iterators add support for this feature. Iterators form a generic interface between algorithms and data structures by abstracting data traversal, access permission, and access patterns. Glift extends the iterator concepts from Boost and the STL to abstract traversal over GPU data structures.

Glift iterators provide the following benefits:

—GPU and CPU iteration over complex data structures;
—generalization of the GPU computation model; and
—encapsulation of GPU optimizations.

Glift supports two types of iterators: element iterators and address iterators. Element iterators are STL-like iterators whose value is a data structure element. In contrast, address iterators are lower-level constructs that traverse the virtual or physical N-D *addresses* of a structure rather then its elements. Most programmers will prefer the higher-level element iterators, but address iterators are important for algorithms that perform computation on virtual addresses or for users wanting to adopt Glift at a lower abstraction level. Section 8 provides a further discussion of Glift iterators, especially the relationship between Glift's iteration model and the stream programming model popularized by Brook and Sh.

2.2.5 *Container Adaptors.* In addition to the four main components, Glift defines higher level data structures as container adaptors. Container adaptors implement their behavior on top of an existing container. For example, in the STL, stacks are container adaptors built atop either a vector or a queue. Section 3 describes a simple N-D array container adaptor, and Section 6 describes two new data structures built as container adaptors: a GPU stack built atop a Glift array and a generic quadtree/octree structure built atop the Glift page-table based address translator.

3. GLIFT PROGRAMMING

The Glift programming model is designed to be familiar to STL, OpenGL, and Cg programmers. In this section, we use the example of building a 4D array in Glift to describe the declaration, use, and capabilities of each Glift component.

Each Glift component defines both the C++ and Cg source code required to represent it on the CPU and GPU. The final source code for a Glift structure is generated by the composition of templated components. Each component is designed to be both composited together or used as a standalone construct. Additionally, all Glift components have a CPU-based version to facilitate debugging and enable algorithms for both the CPU and GPU.

We begin with a complete example, then break it down into its separate components. The C++ and Cg code for defining, instantiating, initializing, binding, and using the 4D Glift array is shown below:

```
typedef glift::ArrayGpu<vec4i, vec4f> ArrayType;

// Instantiate Glift shader parameter
GliftType arrayTypeCg = glift::cgGetTemplateType<ArrayType>();
glift::cgInstantiateNamedParameter(prog, "array", arrayTypeCg);

// ...  compile and load shader ...

// Create Glift 4D array
vec4i size( 10, 10, 10, 10 );
ArrayType array( size );

// Initialize all elements to random values
std::vector<vec4f> data( 10000 );
for (size_t i = 0; i < data.size(); ++i) {
   data[i] = vec4f(drand48(), drand48(), drand48(), drand48());
}
array.write(0, size, data);
```

```
// Bind array to shader
CGparameter param = cgGetNamedParameter(prog, "array");
array.bind_for_read( param );

// ...  Bind shader and render ...
```

The type of the array is declared to be addressed by 4D integers and store 4D float values[1]. Glift adds template support to Cg and so the `array` type of the parameter to the Cg shader must be instantiated with `cgInstantiateParameter`. Next, the array is defined and initialized; then lastly, the array is bound as a shader argument using `bind_for_read`. This is analogous to binding a texture to a shader, but binds the entire Glift structure.

The Cg shader that reads from the 4D Glift array is as follows:

```
float4 main( uniform VMem4D array, varying float4 coord ) : COLOR {
   return array.vTex4D( coord );
}
```

This is standard Cg syntax except the array is declared as an abstract 4D type and `array.vTex4D` replaces native Cg texture accesses. Note that all Glift components are defined in the C++ namespace `glift`; however, all subsequent code examples in this paper exclude explicit namespace scoping for brevity.

3.1 Physical Memory

The `PhysMem` component encapsulates a GPU texture. It is most often instantiated by specifying an address type and value type. These address and value types are statically-sized, multidimensional vectors. For example, the `PhysMem` component for the 4D array example is declared to use 2D integer addressing (`vec2i`) and store 4D float values (`vec4f`) as follows:

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

Just as the STL containers have many default template parameters that are effectively hidden from most users, Glift components have optional template parameters for advanced users. If left unspecified, Glift uses type inference or default values for these extra parameters. In addition to the address and value type, `PhysMem` components are also parameterized by an addressing mode and internal format. The addressing mode is either scaled (integer) or normalized (floating point) and is specified using a Glift type tag[2]. The internal format is the OpenGL enumerant for the internal texture format. The complete template specification for a `PhysMem` component is:

```
typedef PhysMemGPU<AddrType, ValueType,
                   AddrModeTag, InternalFormat> PhysMemType;
```

In the 4D array example, type inference determines the addressing mode to be `ScaledAddressTag` and the internal format to be GL_RGBAF32_ARB.

3.2 Virtual Memory

In the 4D array example, the virtual domain is four dimensional and the physical domain is two dimensional. The generic `VirtMem` Glift component is parameterized by only the physical memory and address translator types:

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

[1]Note that the C++ code in this paper uses vector types such as `vec4f` to indicate statically-sized tuples of either integers or floats. In this case, `vec4f` indicates a tuple of four floats. In the future, Glift will accept user-defined vector/tuple classes that conform to a minimal interface.
[2]Type tags are like C/C++ enumerants, but are more flexible and scalable for use in generic libraries.

The memory copy operations (`read`, `write`, `copy_from_framebuffer`) are specified in terms of contiguous regions of the virtual domain. For example, in the 4D array example, the user reads and writes 4D sub-regions. The `VirtMem` class automatically converts the virtual region to a set of contiguous physical regions and performs the copies on these physical blocks.

3.3 Address Translator

An address translator defines the core of the Glift data structure. They are primarily used to define a `VirtMem` component, however, they may also be used independently as first-class Glift objects to facilitate incremental adoption. Address translators are minimally parameterized by two types: the virtual and physical address types. For example, the address translator for the 4D array example is defined as:

```
typedef NdTo2dAddrTransGPU<vec4i, vec2i> AddrTransType;
```

This typedef defines a 4D-to-2D address translator where both the virtual and physical domain use scaled addressing and no boundary conditions are applied.

For advanced users, address translators are parameterized by at least three additional type: the virtual boundary condition, the virtual addressing mode, and the physical addressing mode. The complete template prototype for an address translator is:

```
typedef AddrTrans<VirtAddrType, PhysAddrType, VirtBoundaryTag,
                  VirtAddrModeTag, PhysAddrModeTag> AddrTransType;
```

The `VirtBoundaryTag` is a Glift type tag defining the boundary conditions to be applied to virtual addresses. It defaults to no boundary condition, but Glift supports all OpenGL wrap modes.

To create a new Glift address translator, users must define the two core translation functions:

```
pa_type translate( const va_type& va );
void    translate_range( const va_type& origin, const va_type& size,
                         range_type& ranges );
```

The `translate` method maps a point in the virtual address space to a point in the physical address space. The `translate_range` method converts a contiguous region of the virtual domain into a set of contiguous physical ranges. Range translation is required for efficiency; it enables Glift to translate all addresses in the range using a small number of point translations followed by interpolation. The `VirtMem` component uses these two methods to automatically create a virtualized GPU memory interface for any randomly-indexable structure.

3.4 Iterators

Iterators are a generalization of C/C++ pointers and abstract data structure traversal, access permissions, and access patterns. This section introduces the use and capabilities of Glift's CPU and GPU element and address iterators. We begin with simple CPU examples and build up to GPU iterators.

3.4.1 *Traversal.* Iterators encapsulate traversal by enumerating data elements into a 1D, linear address space. With iterators, programmers can express per-element algorithms over any data structure. For example, a Glift programmer writes a CPU-based algorithm that negates all elements in the example 4D array as:

```
ArrayType::iterator it;
for (it = array.begin(); it != array.end(); ++it) {
   *it = -(*it);
}
```

where `++it` advances the iterator to the next element, `*it` retrieves the data value, `begin` obtains an iterator to the first element, and `end` returns an iterator one increment past the end of the array.

Glift programmers may also traverse sub-regions of structures by specifying the origin and size of a *range*. For example, to traverse the elements between virtual addresses `(0,0,0,0)` and `(4,4,4,4)`, the programmer writes:

```
ArrayType::range r = array.range( vec4i(0,0,0,0), vec4i(5,5,5,5) );
ArrayType::iterator it;
for (it = r.begin(); it != r.end(); ++it) {
   *it = -(*it);
}
```

Given that the goal of Glift is primarily parallel execution, we need to replace the explicit `for` loop with an encapsulated traversal function that can be parallelized. For example, the STL's `transform` construct can express the previous example as:

```
ArrayType::range r = array.range( vec4i(0,0,0,0), vec4i(5,5,5,5) );
std::transform( r.begin(), r.end(), r.begin(), std::negate<vec4f>() );
```

where the first two arguments specify the input iterators, the third specifies the corresponding output iterator, and the fourth argument is the computation to perform on each element. In stream programming terminology, this fourth argument is called the *kernel*.

While this example is executed serially on the CPU, the `transform` operation's order-independent semantic makes it trivially parallelizeable. Glift leverages this insight to express GPU iteration as parallel traversal over a range of elements.

3.4.2 *Access Permissions.* In addition to defining data traversal, Glift iterators also express data access permissions. Access permissions control if the value of an iterator is read-only, write-only, or read-write. For simplicity, the examples so far have used iterators with read-write access permissions. It is especially important to distinguish between access permissions on current GPUs, which prohibit kernels from reading and writing to the same memory location. The previous example can be re-written to obey such rules by using separate input and output arrays:

```
// ...  Declare arrayIn and arrayOut as identically sized arrays ...

ArrayType::range r = arrayIn.range( vec4i(0,0,0,0), vec4i(5,5,5,5) );
std::transform( r.begin_i(), r.end_i(), arrayOut.begin_o(), std::negate<vec4f>() );
```

Note that `begin_i` and `end_i` return read-only (input) iterators and `begin_o` returns a write-only output iterator. One-sided communication models such as this avoid synchronization problems and are also known to be very efficient on CPU-based parallel computers [Kendall et al. 2005].

3.4.3 *Access Patterns.* Glift iterators support three types of data access patterns: single, neighborhood, and random. An iterator's access pattern provides a mechanism by which programmers can explicitly declare their application's memory access patterns. In turn, this information can enable compilers or runtime systems to hide memory access latency by pre-loading data from memory before it is needed. As noted in Purcell et al. [2002], future architectures could use this information to optimize cache usage based on the declared memory access pattern. For example, the stream model permits only single-element access in kernels, enabling perfect pre-loading of data before it is needed. At the other extreme, random memory accesses afford little opportunities to anticipate which memory will be needed. Single-access iterators do not permit indexing and are analogous to Brook's stream inputs and the STL's non-random-access iterators. Neighborhood iterators permit relative indexing in a small, constant-sized region around the current element. These iterators are especially useful in image processing and grid-based simulation applications. Random-access iterators permit indexing into any portion of the data structure.

3.4.4 *GPU Iterators.* We now have enough machinery to express a GPU version of the CPU example shown in Section 3.4.1. To begin, the Cg source for `negateKernel` is:

```
void main( SingleIter it, out float4 result : COLOR ) {
   result = -(it.value());
}.
```

`SingleIter` is a Glift Cg type for a single-element (stream) iterator and `it.value()` dereferences the iterator to obtain the data value.

The C++ source that initiates the execution of the GPU kernel is:

```
ArrayType::gpu_in_range  inR = array1.gpu_in_range( vec4i(0,0,0,0), vec4i(5,5,5,5) );
ArrayType::gpu_out_range outR = array2.gpu_out_range( vec4i(0,0,0,0), vec4i(5,5,5,5) );

CGparameter arrayParam = cgGetNamedParameter(prog, "it");
inR.bind_for_read( arrayParam );
outR.bind_for_write( GL_COLOR_ATTACHMENT0_EXT );

// ...  Bind negateKernel fragment program ...

exec_gpu_iterators(inR, outR);
```

The `gpu_in_range` and `gpu_out_range` methods create entities that specify GPU computation over a range of elements and deliver the values to the kernel via Cg iterators. Note that GPU range iterators are first-class Glift primitives that are bound to shaders in the same way as Glift data structures.

The `exec_gpu_iterators` call is a proof-of-concept GPU execution engine that executes `negateKernel` across the specified input and output iterators but is not a part of core Glift. The goal of Glift is to provide generic GPU data structures and iterators, not provide a runtime GPU execution environment. Glift's iteration mechanism is designed to be a powerful back-end to GPU execution environments such as Brook, Scout, or Sh.

3.4.5 *Address Iterators.* In addition to the data element iterators already described, Glift also supports *address iterators*. Rather than iterate over the values stored in a data structure, address iterators traverse N-D virtual or physical addresses. Address iterators enable users to specify iteration using only an `AddrTrans` component.

A simple CPU example using an address iterator to add one to all elements of the example 4D array is:

```
AddrTransType::range    r = addrTrans.range( vec4i(0,0,0,0), vec4i(5,5,5,5) );
AddrTransType::iterator ait;

for (ait = rit.begin(); ait != rit.end(); ++ait) {
   vec4i va = *ait;
   vec4f val = array.read( va );
   array.write( va, val + 1 );
}
```

Note that iteration is now defined with an address translator rather than a virtual or physical container, and the value of the iterator is an index rather than a data value.

The Cg code for a GPU address iterator example is:

```
float4 main( uniform VMem4D array, AddrIter4D it ) : COLOR {
   float4 va = it.value();
   return array.vTex4D(va);
}
```

and the corresponding C++ code is:

```
AddrTransType::gpu_range r = addrTrans.gpu_range( vec4i(0,0,0,0), vec4i(5,5,5,5) );

// ...  Bind Glift parameters ...
// ...  Bind Cg shader ...

exec_gpu_iterators(r);
```

3.5 Container Adaptors

Container adaptors are higher-level containers that define a behavior atop an existing `VirtMem` structure. For example, the `ArrayGpu` template class shown in the example at the beginning of Section 3 is a simple container adaptor built atop the set of typedefs developed in the preceding sections:

```
typedef PhysMemGPU<vec2i, vec4f>                PhysMemType;
typedef NdTo2dAddrTransGPU<vec4i, vec2i>        AddrTransType;
typedef VirtMemGPU<PhysMemType, AddrTransType> VirtMemType;
```

The `ArrayGpu` container adaptor encapsulates this definition such that users can declare an array simply as:

```
typedef glift::ArrayGpu<vec4i, vec4f> ArrayType;
```

Section 6 describes more complex container adaptors.

4. GLIFT DESIGN AND IMPLEMENTATION

The Glift C++ template library maps the abstractions described in Section 2 to a C++/OpenGL/Cg GPU programming environment. This section describes the design and implementation of Glift, including how Glift unifies the CPU and GPU source code, adds template support to Cg, and virtualizes the multi-processor, multi-language GPU memory model.

To support the design goals of the abstraction presented in Section 2, the implementation of Glift is designed with the following goals:

—incremental adoption;
—extensibility;
—efficiency; and
—CPU and GPU interoperability.

Glift supports easy, incremental adoptability by providing familiar texture-like and STL-like interfaces, requiring minimal changes to shader authoring and compilation pipelines, and allowing Glift components to be used alone or in combination with others. Glift offers easy extensibility by providing three ways to create new, fully-virtualized Glift data structures: write a new address translator, change the behavior of an existing Glift component by writing a new policy[3], or write a container adaptor that maps new behavior atop an existing structure. Glift's efficiency mechanisms include static polymorphism, template specialization, program specialization, and leveraging optimizing GPU compilers. Lastly, Glift supports processor interoperability by supporting CPU and GPU memory mappings and iterators.

[3]Policy-based template design factors classes into orthogonal components, making it possible to change behavior by replacing one small module [Alexandrescu 2001]

4.1 GPU Code Generation and Compilation

This section describes how Glift adds template-like support to a GPU shading language. We chose to make Glift a C++ template library based on the goal to provide a high-performance programming abstraction. High-performance C++ libraries such the STL, Boost, and POOMA [Karmesin et al. 2002] have demonstrated the power of static polymorphism for providing abstractions for performance-critical coding. The challenge, however, is that current GPU languages do not support templates.

We began by selecting Cg as the target GPU language. This was largely based on its support for primitive static polymorphism via its *interface* construct [Pharr 2004]. Interfaces allow shader writers to specify input parameter types in terms of an abstract interface whose concrete type is determined at compile time.

Given that the GPU shading code could not be templatized and our desire for Glift structures to support CPU and GPU usage, we integrated the GPU and CPU code bases. Each Glift component contains templatized C++ code and stringified GPU code. This design enables us to generate GPU code from C++ template instantiations. This generated Cg code becomes the concrete implementation of the interface type declared by a shader.

The goals for integrating templates into Cg include easy integration with existing Cg shader compilation work flow and minimal API extensions to the Cg API. The resulting system adds only two new Cg API calls:

```
GliftType cgGetTemplateType<GliftCPlusPlusType>();
CGprogram cgInstantiateParameter(CGprogram, const char*, GliftType);
```

The first call maps a C++ type definition to a runtime identifier (`GliftType`). This identifier is then passed to the second new call, `cgInstantiateParameter`, to insert the Glift source code into the shader. The first argument is the Cg program handle, the second is the name of the abstract interface parameter this Glift type will instantiate, and the third is the `GliftType` identifier. The call prepends the Glift Cg code into the shader and defines the `GliftType` to be the concrete implementation of the interface shader parameter. The returned Cg program is compiled and loaded like a standard Cg shader. The only other change to the shader pipeline is that Glift parameters are bound to shaders by calling `bind_for_read` instead of using Cg's standard parameter value setting routines.

4.1.1 *Alternate Designs.* The above design was decided upon after multiple early design iterations. One of these earlier approaches obtained the Glift Cg source code from instances of Glift data structures. This approach proved burdensome and clumsy because it required the Glift data structure to be present and initialized at shader compile time. The final model instead requires only the `GliftType` identifier. This identifier can be obtained either from the C++ type definition or a Glift object.

4.2 Cg Program Specialization

Glift leverages Cg's *program specialization* capability to help create efficient code from generic implementations. Program specialization is a program transformation that takes a procedure and some of its arguments, and returns a procedure that is the special case obtained by fixing those arguments [Guenter et al. 1995]. The Cg API supports specialization on shaders by having users set the value of uniform parameters, change their variability to constant using `cgSetParameterVariability`, then recompile the program to generate a specialized version of it.

Glift components support specialization by providing an analogous `set_member_variability` method. This call specializes (or un-specializes) all uniform parameters defined by the component. This feature allows users to "bake out" many parameters that will not change at run time at the cost of additional compilation time and shader management.

4.3 Iterators

As described in Section 2.2.4, Glift supports two kinds of iterators: address iterators and element iterators. Here, we describe how these constructs map to current graphics hardware.

4.3.1 *Address Iterators.* Address iterators are an important concept for the Glift implementation for two reasons. First, address iterators can be efficiently implemented as GPU rasterizer interpolants (i.e., texture coordinates). Second, they are the iterator type supported by `AddrTrans` components. The concept enables address translators to specify iteration without having knowledge of physical data.

Glift address translators generate a stream of valid virtual addresses for a GPU data structure. They abstract the screen-space geometry used to initiate GPU computation as well as the physical-to-virtual translation required to map from fragment position to the virtual address domain. Glift represents address iterators using vertex data, vertex shader code and parameters, and a viewport specification.

Note that this design means that modifying a Glift data structure (e.g., inserting or deleting elements) means writing to both the address translator memory and the iterator representation. The latter operation requires either render-to-vertex-array support or CPU involvement.

4.3.2 *Element Iterators.* Glift element iterators pose several challenges for implementation on current GPUs. First, element iterators are a pointer-like abstraction, and GPUs do not support pointers. Second, GPU data structure traversal is usually accomplished with the rasterizer, yet the rasterizer cannot generate true memory addresses. It can only generate N-D indices via texture coordinate interpolation (i.e., address iterators). As such, Glift implements element iterators by combining an address iterator with a Glift container (either `PhysMem` or `VirtMem`). The last challenge is the design requires shading language structures that contain a mixture of `uniform` and `varying` members. This feature has recently only been added to Cg and is not yet supported in other languages.

4.4 Mapping Glift Data into CPU Memory

All Glift components are designed to support data access on both the CPU and GPU. Glift implements this feature by supporting explicit `map_cpu` and `unmap_cpu` functionality. Just as with mappable GPU memory like vertex buffer objects and pixel buffer objects, `map` copies the data into CPU memory while `unmap` copies data into GPU memory. The default behavior is to transfer the entire data structure; however, Glift can optionally operate in a paged, lazy-transfer mode where only the required portions of the structure are copied. This mode reduces CPU read/write efficiency but can greatly reduce the amount of data sent between the CPU and GPU.

Initial designs attempted to automatically map data to the appropriate processor based on usage. This proved problematic, however, because it is not possible to detect all cases when a GPU-to-CPU or CPU-to-GPU synchronization is required. For example, this can arise if a user binds a Glift structure to a shader, writes to the CPU mapping of the structure, then later re-binds the shader without explicitly re-binding the Glift structure. This is a perfectly legal OpenGL usage pattern, but one in which Glift cannot detect that the structure is now being read by the GPU. To support this automatic virtualization, Glift would either have to subsume a much larger portion of the GPU programming model (shader binding, etc.) or be integrated directly into GPU drivers.

4.5 Virtualized Range Operations

Providing generic support for virtualized range operations (read, write, copy) was one of the toughest challenges in the design of Glift. The goal is for application programmers to be able to easily create new containers that have full support for these operations. The challenge is that the contiguous virtual region over which these operations are defined often maps to multiple physical regions.

Early designs placed this functionality in the `VirtMem` component. This approach, however, required each data structure to implement a significant amount of redundant, complex code. Our solution is inspired by a concept from automatic parallelization research. In their work to create a parallel STL, Austern et al. describe a *range partition adaptor* as an entity that takes a range defined by a `begin` and `end` iterator and breaks it up into subranges which can be executed in parallel [1996]. Applying this idea to Glift, the generic range operation problem is solved if address translators can translate a virtual range into an ordered sequence of physical ranges.

The prototypical usage of this operation is:

```
vec3i virtOrigin(0), virtSize(1,2,7); // input:  origin and size of virtual rectangle
AddrType::ranges_type ranges;         // ouput:  corresponding physical ranges
addrTrans.translate_range( virtOrigin, virtSize, ranges );

for (size_t i = 0; i < ranges.size(); ++i) {
  DoPhysRangeOp( ranges[i].origin, ranges[i].size );    // perform operation on ranges
}
```

where `DoPhysRangeOp` performs an operation such as read/write/copy across a range of contiguous physical addresses. The generic `VirtMem` class template now uses this idiom to virtualize all of the range-based memory operation for *any* address translator.

While conceptually simple, the design had a profound effect on Glift: address translators are now composable. The design places all of the data structure complexity into the address translator, which in turn means that fully virtualized containers can be constructed by combining simpler structures with little or no new coding.

5. CLASSIFICATION OF GPU DATA STRUCTURES

This paper demonstrates the expressiveness of the Glift abstractions in two ways. First, this section characterizes a large number of existing GPU data structures in terms of Glift concepts. Second, Section 6 introduces novel GPU data structures and two applications not previously demonstrated on the GPU due to the complexity of their data structures. The classification presented in this section identifies common patterns in existing work, showing that many structures can be built out of a small set of common components. It also illuminates holes in the existing body of work and trends.

5.1 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. Note that some of the structures in Table II use ND-to-MD mappings as part of more complex structures. These translators are $O(1)$ memory complexity, $O(1)$ access complexity, uniform access consistency, GPU or CPU location, a complete mapping, one-to-one, and invertible.

The mappings 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. C/C++ compilers and Brook support 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. Lefohn et al. [2005] detail the implementation of both of these mappings for current GPUs. Glift currently provides a generic ND-to-2D address translator called `NdTo2dAddrTransGPU`.

5.2 Page Table Translators

Many of the GPU-based sparse or adaptive structures in Table II are based on a page table design. These structures are a natural fit for GPUs, due to their uniform-grid representation; fast, uniform access; and support for dynamic updates. Page tables use a coarse, uniform discretization of a virtual address space to map a subset of it to physical memory. Like the page tables used in the virtual memory system of modern operating systems and microprocessors, page table data structures must efficiently map a block-contiguous, sparsely allocated large virtual address space onto a limited amount of physical memory [Kilburn et al. 1962; Lefohn et al. 2004].

Page table address translators are characterized by $O(N)$ memory complexity, $O(1)$ access complexity, uniform access consistency, GPU or CPU location, complete or sparse mapping, and one-to-one or many-to-one mapping. Page tables are invertible if an inverse page table is also maintained.

Citation	Domains	Mapping		AddrTrans Loc	MCmplx	ACmplx	ACons	Mut?	Notes
[Binotto et al. 2003]	4D→2D	C	A	GPU	n	1	U	Y	2-level page table + hash on time dimension
[Buck et al. 2004]	1D→2D	C	U	GPU	1	1	U	N	Streams: 1D-to-2D mapping
	{1–4}D→2D	C	U	GPU	1	1	U	N	Arrays: ND-to-1D-to-2D mapping
[Carr and Hart 2004]	3D→2D	S	A	CPU	$\log n$	$\log n$	NU	Y	Quad-tree on CPU
[Coombe et al. 2004]	3D→2D	C	A	CPU	$\log n$	$\log n$	NU	Y	Quad-tree on CPU
[Foley and Sugerman 2005]	3D→2D	C	A	GPU	$\log n$	$\log n$	NU	N	Static k-d tree on GPU
[Harris et al. 2003]	3D→2D	C	U	GPU	1	1	U	N	3D-to-2D page-based
[Johnson et al. 2005]	2D→2D	S	A	GPU⋆	n	$\propto$ list size	NU	Y	1-level page table
[Kraus and Ertl 2002]	{2,3,4}D→{2,3}D	S	A	GPU	n	1	U	N	1-level page table
[Lefebvre et al. 2004]	3D→3D	S	A	GPU	$\log n$	$\log n$	NU	N	Variable-level 3D page table
[Lefohn et al. 2004]	3D→2D	S	U	CPU	n	1	U	Y	1-level 3D page, 2D pages
[Purcell et al. 2002]	3D→2D	C	U	GPU	n	$\propto$ list size	NU	Y	1-level 3D page table. Grid of lists
[Purcell et al. 2003]	3D→2D	C	U	GPU	n	$\propto$ list size	NU	Y	1-level 3D page table. Grid of lists
[Schneider and Westermann 2003]	3D→3D	C	A	GPU	n	1	U	N	1-level page table, 2 VQ code book lookups, decompression math
[Sen 2004]	2D→2D	C	A	GPU	n	1	U	Y	Non-linear warp map to deform uniform physical memory to represent an adaptive virtual space
[Tarini et al. 2004]	3D→2D	S	U	GPU	n	1	U	N	1-level 3D page table, 3D-to-3dPolyCube, 3dPolyCube-to-2D
[Thrane and Simonsen 2005]	3D→2D	C	A	GPU	$\log n$	$\log n$	NU	N	Static GPU bounding volume hierarchy

Table II: Characterization of previous GPU data structures expressed in terms of address translator characteristics. The purpose of this table is to identify similarities between structures and illuminate gaps in the existing body of research. For each work, we describe the virtual and physical address domains and the six characteristics described in Section 2.2.3. Mappings are described as complete or sparse (C/S) and uniform or adaptive (U/A). The address translation column indicates if the implementation uses the CPU or the GPU to perform the address translation (Loc), the memory complexity (MCmplx), the access complexity (ACmplx), the access consistency (ACons), and if the structure is updatable on the GPU (Mut). The last column provides a high-level description of the data structure. ⋆ indicates "proposed".

Page tables have the disadvantage of requiring $O(N)$ memory, where N is the size of the virtual address space; however, multilevel page tables can dramatically reduce the required memory. In fact, the multilevel page table idiom provides a continuum of structures from a 1-level page table to a full tree. This was explored by Lefebvre et al. [2004], who began with a full tree structure and found the best performance was with a shallower-branching multilevel page table structure. Section 6.3 describes a new dynamic, multiresolution, adaptive GPU data structure that is based on a page table design.

The basic address translation calculation for an N-D page table translator is shown below:

```
vpn = va / pageSize              // va:  virtual address
                                 // vpn:  virtual page number
pte = pageTable.read( vpn )      // pte:  page table entry
```

```
ppn = pte.ppn()                 // ppn:  physical page number
ppa = ppn * pageSize            // ppa:  physical page origin
off = va % pageSize             // off:  offset into physical page
pa  = ppa + off                 // pa :  physical address
```

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. Glift provides a generic page table address translator. The policy-based implementation is highly parameterized to maximize code reuse. As demonstrated in Section 6, entirely new data structures can by created by writing a small amount of policy code for the existing class template.

5.3 GPU Tree Structures

Until 2004, nearly all GPU-based address translators were $O(1)$ access complexity and uniform consistency. In the last two years, however, researchers have begun to implement tree structures such as k-d trees, bounding volume hierarchies, and N-trees. One notable precursor to these recent structures is Purcell's non-uniform consistency grid-of-lists construct used to build a ray tracing acceleration structure and kNN-grid photon map in 2002 and 2003, respectively.

The change to non-uniform access consistency structures came largely as a result of NVIDIA releasing the NV40 architecture with support for looping in fragment programs. Support for this feature is, however, still primitive and incoherent branching can greatly impact performance [Harris and Buck 2005]. Purcell and Foley both avoided the need for fragment program looping by rendering one loop iteration per pass, but this approach significantly increases the bandwidth requirement for data structure traversal. As future GPUs provide more efficient branching support, these memory-efficient data structures will continue to improve in performance.

5.4 Dynamic GPU Structures

Including dynamic complex data structure in GPU applications is an area of active research. In fact, Purcell et al. are the only researchers to describe an entirely GPU-updated, dynamic sparse data structure. Lefohn et al. [2004] and Coombe et al. [2004] both describe efficient GPU-based dynamic algorithms that use the CPU only as a memory manager. Clearly, GPU-updated sparse and adaptive structures will be an area of focus for the next couple of years, especially with the interest in ray tracing of dynamic scenes.

Section 6 introduces a new GPU-based dynamic sparse and adaptive structure. One of the conclusions from that section is that an efficient *stream compaction* algorithm is required to build structures with parallel computation [Horn 2005]. This conclusion is also noted in the parallel computing literature [Hoel and Samet 1995].

5.5 Limitations of the Abstraction

Most of the structures listed in Table II are encapsulated well by the Glift abstractions. One structure that is challenging to express in Glift is the grid-of-lists used by Purcell et al. [2003] and Johnson et al [2005]. This structure is a page table with variable-size pages. Interpreting this as a random-access container requires using a 1D index for elements in the variable-sized pages; thus a 3D grid-of-lists uses a 4D virtual address space.

A second challenge to the Glift abstraction is the interface for building each of the structures. The sparse and adaptive structures, in particular, require additional methods to allocate and free memory. In practice, we have added these operations in the high-level container adaptor classes. In the future, we may be able to identify a small set of generic interfaces that abstract these operations.

6. CASE STUDIES

This section describes how Glift is used to build three GPU data structures, two of which are novel. It also demonstrates the increased program complexity made possible by Glift by describing two high-quality interactive rendering

applications of our multiresolution, adaptive data structure: adaptive shadow maps and octree 3D paint.

6.1 GPGPU 4D Array

After describing the Glift components required to build and use a 4D array in Section 2, we now show a complete "before" and "after" transformation of source code with Glift. We use a GPGPU computation example because of its simplicity; note that it demonstrates the use of Glift element iterators. Section 6.3 shows the use of Glift for graphics GPU applications.

Here, we show the Cg code for the Glift and non-Glift versions of this example. Appendix 1 shows the corresponding C++ code for both examples. For each element in the 4D array, the following kernel computes the finite discrete Laplacian. The Cg shader for the non-Glift example is:

```
float4 physToVirt( float2 pa, float2 physSize, float4 sizeConst4D )
{
   float3 curAddr4D;
   float  addr1D = pa.y * physSize.x + pa.x;
   addr4D.w  = floor( addr1D / sizeConst4D.w );
   addr1D   -= addr4D.w * sizeConst4D.w;
   addr4D.z  = floor( addr1D / sizeConst4D.z );
   addr1D   -= addr3D.z * sizeConst4D.z;
   addr4D.y  = floor( addr1D / sizeConst4D.y );
   addr4D.x  = addr1D - addr4D.y * sizeConst4D.y;
   return addr4D;
}

float2 virtToPhys( float4 va, float2 physSize, float4 sizeConst4D )
{
   float  addr1D = dot( va, sizeConst4D );
   float  normAddr1D = addr1D / physSize.x;
   return float2( frac( normAddr1D ) * physSize.x, normAddr1D );
}

float4 main( uniform sampler2D array1,
             uniform float4    virtSize,
             uniform float2    physSize,
             uniform float4    sizeConst,
             varying float2    winPos : WPOS ) : COLOR
{
   // Get virtual address for current fragment
   float2 pa = floor( winPos );
   float4 va = physToVirt( pa, physSize, sizeConst );

   // Finite difference discrete Laplacian
   float4 offset( 1, 0, 0,  0);
   float4 laplace = -8 * tex2D( array1, pa );
   for( float i = 0; i < 4; ++i ) {
      laplace += tex2D( array1, virtToPhys( va + offset, physSize, sizeConst ) );
      laplace += tex2D( array1, virtToPhys( va - offset, physSize, sizeConst ) );
      offset = offset.yzwx;
   }
   return laplace;
```

```
}
```

Note that this Cg program includes both a physical-to-virtual and virtual-to-physical address translation. The former maps the output fragment position to the virtual domain of the 4D array, and the latter maps the virtual array addresses to physical memory. While the functions encapsulate the address translation, the data structure details obscure the algorithm, and the shader is hard-coded for this particular 4D array. Any changes to the structure require rewriting the shader.

In contrast, the Cg shader for the Glift version of the same example is:

```
#include <gliftCg.h>

float4 main( NeighborIter it ) : COLOR
{
   // Finite difference discrete Laplacian
   float4 offset( 1, 0, 0, 0 );
   float4 laplace = -8 * it.value( 0 );
   for( float i = 0; i < 4; ++i ) {
      laplace += it.value( offset );
      laplace += it.value( -offset );
      offset = offset.yzwx;
   }
   return laplace;
}
```

The `NeighborIter` parameter is a neighborhood iterator that gives the kernel access to a limited window of data values surrounding the current stream element. Note the intent of the algorithm is much clearer here than in the non-Glift version, the code is much smaller, and the algorithm is defined completely separately from the data structure. The only requirement of the data structure is that it supports neighbor iterators with 4D offsets.

There is an additional subtle benefit to the Glift version of the code. The expensive address translation shown in the non-Glift Cg shader can be optimized without changing the C++ or Cg user code. The iterator can pre-compute the physical-to-virtual and virtual-to-physical address translations. These optimizations can be performed using pre-computed texture coordinates, the vertex processor, and/or the rasterizer. These types of optimization have been performed by hand in numerous GPGPU research papers [Bolz et al. 2003; Goodnight et al. 2003; Govindaraju et al. 2005; Lefohn et al. 2004; Strzodka and Telea 2004], and Section 8 discusses this important property of iterators in more detail.

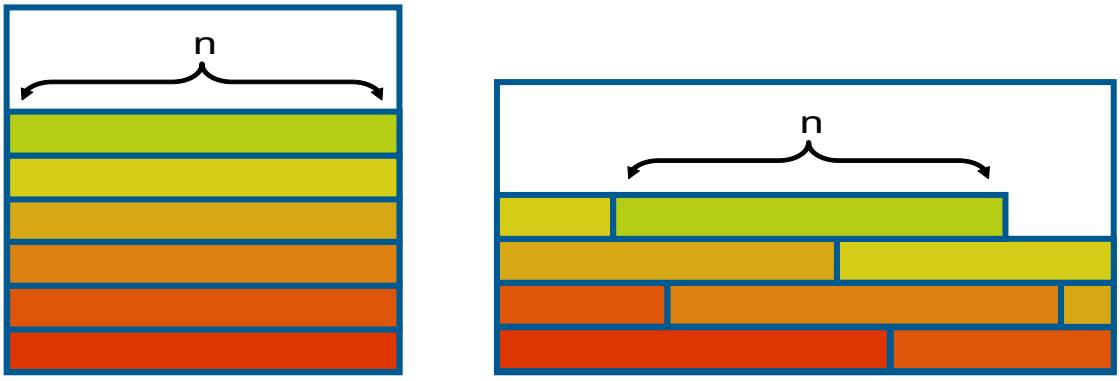
6.2 GPU Stack

The stack is a fundamental data structure in CPU programming, yet there has been little research on a GPU version of it. Applications of a GPU stack for graphics structures include k-d tree traversal for ray tracing and GPU-based memory allocation for page-table structures (see Section 6.3). This section introduces a GPU-compatible stack-like structure called the n-stack. The structure is a more general version of the *multistack* presented in work performed simultaneously with ours [Ernst et al. 2004]. It is implemented as a Glift container adaptor atop a virtualized 1D array.

The n-stack can be thought of as n stacks combined into a single structure (see Figure 3). Each `push` and `pop` operation processes n elements in parallel. We implement the n-stack as a varying-size 1D array with a fixed maximum size.

A Glift n-stack of 4-component floats with `n = 1000` is declared as:

```
const int n = 1000;
typedef StackGPU<n, vec4f> StackType;
```



(a) (b)

Fig. 3: The Glift *n*-stack container stores a stack of *n*-length arrays (a) to support *n*-way parallel push and pop operations. We implement the structure in Glift as a 1D virtual array stored in 2D physical memory (b).

```
int maxNumN = 10000;
StackType stack( maxNumN );
```

The stack can store at most `maxNumN` number of *n*-sized arrays. Users may push either CPU-based or GPU-based data onto the stack. The following code demonstrates a GPU stack push:

```
typedef ArrayGpu<vec1i, vec4f> ArrayType;
ArrayType data( 1000 );

// ...  initialize data array ...

ArrayType::gpu_in_range r = data.gpu_in_single_range( 0, 1000 );
stack.push( r );
```

Push writes a stream of *n* input values into the physical memory locations from `stack.top()` to `stack.top() + n - 1` (see Figure 4). Note that the input to `push` is specified as a range of element iterators rather than a specific data structure. This makes `push` compatible with any Glift data structure rather than just dense, 1D arrays.

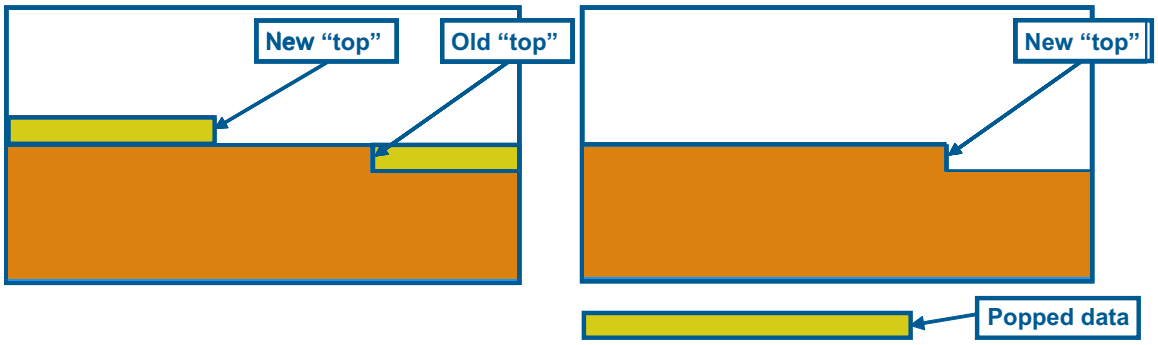
Pop removes the top *n* elements from the stack and returns a GPU input range iterator (Figure 4). This iterator can be bound as input to a GPU computation over the specified range. Example C++ source code that pops two streams from the stack, combines them in a subsequent kernel, and writes them to an array looks like:

```
StackType::gpu_in_range result1 = stack.pop();
StackType::gpu_in_range result2 = stack.pop();

// ...  Create Cg shader and instantiate Glift types

CGparameter result1Param = cgGetNamedParameter( prog, "result1" );
result1.bind_for_read( result1Param );

CGparameter result2Param = cgGetNamedParameter( prog, "result2" );
```



(a) (b)

Fig. 4: The Glift *n*-stack container supports pushing and popping of *n* elements simultaneously. The push operation writes elements from positions `top` to `top + n - 1` (a). The pop operation returns a range iterator that points to elements from `top` to `top - n`, then decrements `top` by `n` (b).

```
result2.bind_for_read( result2Param );

ArrayType output( StackType::value_size );
ArrayType::gpu_out_range outR = output.gpu_out_range( 0, StackType::value_size );
outR.bind_for_write( GL_COLOR_ATTACHMENT0_EXT );

glift::execute_gpu_iterator( result1, result2, outR );
```

The corresponding Cg shader is:

```
float4 main( SingleIter result1, SingleIter result2 ) : COLOR {
   return result1.value() + result2.value();
}
```

Our current implementation does not support a Cg interface for `push` or `pop` because these operations modify the stack data structure. Current GPUs do not support read-write memory access within a rendering pass, and so `push` must execute in its render pass. Note that `pop` returns only a GPU input iterator and therefore does not require its own render pass.

6.3 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 $524{,}288^2$ adaptive shadow map that consumes only 16 MB of RAM and a 1024^3 fluid simulation grid. Unfortunately, the complexity of adaptive-grid data structures (usually tree structures) has, for the most part, has prevented their use in real-time graphics and GPU-based simulations that require a dynamic adaptive representation.

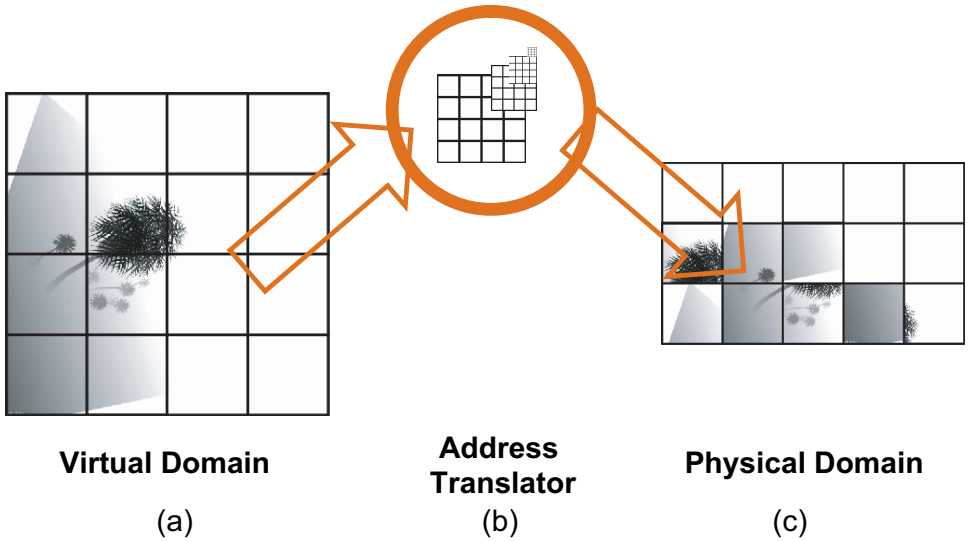


Fig. 5: This figure shows the three components of the multiresolution, adaptive structure used to represent a quadtree and octree. In this adaptive shadow mapping example, the virtual domain (a) is formed by the shadow map coordinates. The address translator (b) is a mipmap hierarchy of page tables, and the physical memory (c) is a 2D buffer that stores identically sized memory pages. The address translator supports adaptivity by mapping a varying number of virtual pages to a single physical page.

Previous work on adaptive GPU data structures (see Table II) includes CPU-based address translators [Carr and Hart 2004; Coombe et al. 2004], static GPU-based address translators [Binotto et al. 2003; Foley and Sugerman 2005; Kraus and Ertl 2002; Thrane and Simonsen 2005], 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 quadlinear [3D] filtering). Section 6.5 describes the differences between our structure and that of work done in parallel with ours by Lefebvre et al. [2004].

6.3.1 *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 5.2. 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 6.2 defines a stack on top of a 1D virtual memory definition). The structure supports adaptivity via a many-to-one address translator that can map multiple virtual pages to the same physical page. We avoid the bin-packing problem and enable mipmap filtering by making all physical pages the same size. We support multiresolution via a mipmap hierarchy of page tables. Figure 5 shows a diagram of our structure.

The adaptive structure is predominantly built out of generic Glift components and requires only a small amount of new coding. The new code includes several small policy classes for the generic page table address translator, a page allocator, and the GPU iterators. The resulting `AdaptiveMem` container adaptor is defined as:

```
typedef AdaptiveMem< VirtMemPageTableType, PageAllocator> AdaptiveMemType;
```

`VirtMemPageTable` is the type definition of the generic `VirtMem` component built as a 1-level page table structure. 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 mipmapping. Note that the allocator is a similar construct to the allocators used in the STL to generically support multiple memory models.

The adaptive address translation function differs from the one presented in Section 5.2 in only two small ways. First,

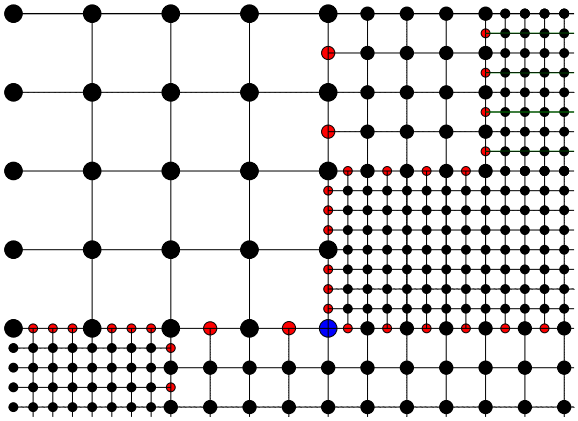


Fig. 6: Depiction of our node-centered adaptive grid representation. Red nodes indicate T-junctions (hanging nodes). The blue node emphasizes the complex situation at the junction of multiple levels of resolution.

we add a mipmap level index to the page table read:

```
pte = pageTable.read( vpn, level )
```

Second, we support variable-sized virtual pages by storing the resolution level of virtual pages in the page table and permitting redundant page table entries (Figure 5). We thus change the offset computation to:

```
off = (va >> pte.level()) % physicalPageSize
```

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) at the cost of using slightly more physical memory. In addition, uniform physical pages simplifies support for dynamic structures by avoiding the bin-packing problem encountered when allocating variable-sized physical pages. See Section 7 for memory system benchmarks that further indicate that complex page packing schemes are unnecessary.

6.3.2 *Adaptivity Implementation Details.* Correct representation of data on an adaptive grid presents several challenges irrespective of its GPU or CPU implementation. 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 values. T-junctions (red nodes in Figure 6) 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.

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

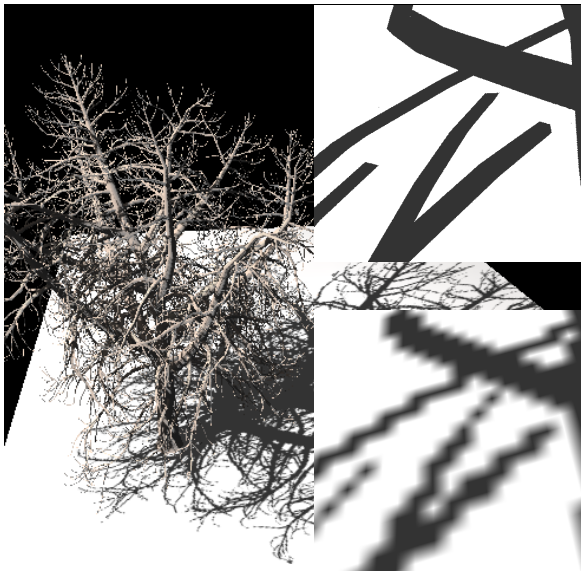


Fig. 7: This adaptive shadow map uses a GPU-based adaptive data structure built with the Glift template library. It has an effective shadow map resolution of $131{,}072^2$ (using 37 MB of GPU memory, trilinearly filtered). The top-right inset shows the ASM and the bottom-right inset shows a 2048^2 standard shadow map.

of data elements on the borders is identical and independent of the resolution (see edge nodes in Figure 6). 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_TO_EDGE` and `GL_REPEAT` boundary modes.

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. This approach has been described in previous literature on adaptive GPU structures [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 are correctly handled by the same interpolation scheme.

6.4 Adaptive Shadow Maps

This section describes the first of two applications of the multiresolution adaptive structure defined in Section 6.3. This case study demonstrates the increased program complexity made possible by Glift by describing the solution to a complex application not previously shown on the GPU. The implementation uses a blend of traditional graphics and GPGPU stream programming.

Adaptive shadow maps [Fernando et al. 2001] offer a rigorous solution to projective and perspective shadow map aliasing while maintaining the simplicity of a purely image-based technique. The complexity of the ASM data structure, however, has prevented full GPU-based implementations until now. We present a novel implementation of adaptive shadow maps (ASMs) that performs all shadow lookups and scene analysis on the GPU, enabling interactive rendering with ASMs while moving both the light and camera. We support shadow map effective resolutions up to $131{,}072^2$ and, unlike previous implementations, provide smooth transitions between resolution levels by trilinearly filtering the shadow lookups (see Figure 7).

Shadow maps, which are 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; Stamminger and Drettakis 2002; Wimmer

et al. 2004]. ASMs nearly eliminate shadow map aliasing by ensuring that the projected area of a screen-space pixel into light space matches the shadow map sample area.

An ASM replaces a traditional shadow map's uniform grid with a quadtree of small shadow map pages (see Figures 5 and 1). Our ASM data structure is an instantiation of the general `AdaptiveMem` Glift structure defined in Section 6.3. Figure 1 shows the factoring of the ASM data structure into Glift components as well as a visualization of the page borders mapped atop a rendered image. Shaders access our ASM structure in the same way as a traditional shadow map lookup. Below is an example of a Cg shader that performs an ASM lookup:

```
float4 main( uniform VMem2D adaptiveShadow, float3 shadowCoord ) : COLOR {
   return adaptiveShadow.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 2×2 percentage-closer filtering (PCF) to return a fractional scalar value indicating how much light reaches the pixel. 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.

The ASM refinement algorithm builds the GPU data structure and proceeds as follows:

```
void refineASM()
{
  AnalyzeScene(...);      // Identify shadow pixels with resolution mismatch
  StreamCompaction(...); // Pack these pixels into small stream
  CpuReadback(...);       // Read refinement request stream
  AllocPages(...);        // Render new page table entries into mipmap page tables
  CreatePages(...);       // Render depth into ASM for each new page
}
```

The algorithm begins by performing a scene analysis to determine which camera-space pixels require refinement. A pixel requires refinement if it lies on a shadow boundary and its required resolution is not in the current ASM. We use a Sobel edge detector to identify shadow boundaries and compute required resolutions using derivatives of the shadow coordinates. We then pack the pixels needing refinement into a small contiguous stream using Horn's stream compaction algorithm [2005]. This small image is read back to the CPU to initiate new shadow data generation. The CPU adds new shadow data to the ASM by first rendering new page allocations into the GPU-based page tables, then rendering the scene geometry from the light into the new physical pages. We repeat this refinement algorithm to convergence.

We tested our ASM implementation on an NVIDIA GeForce 6800 GT using a window size of 512^2. For a 100k polygon model and an effective shadow map resolution of $131{,}072^2$, our implementation achieves 13–16 frames per second while the camera is moving. We achieve 5–10 fps while interactively moving the light for the same model, thus rebuilding the entire ASM each frame.

ASM lookup performance is between 73–91% of a traditional 2048^2 shadow map. 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 for a 512^2 image window. We list results for bilinearly filtered ASM (ASM L), bilinearly filtered mipmapped ASM (ASM LMN), and trilinearly filtered ASM (ASM LML).

PageSize	FPS	ASM L	ASM LMN	ASM LML
8^2	13.7	91%	77%	74%
16^2	15.6	90%	76%	73%
32^2	12.1	89%	75%	73%
64^2	12.9	89%	74%	73%

Fig. 8: Our interactive 3D paint application stores paint in a GPU-based octree-like data structure built with the Glift template library. These images show an 817k polygon model with paint stored in an octree with an effective resolution of 2048^3 (using 15 MB of GPU memory, quadlinear filtered).

The memory consumed by the ASM is configurable and is the sum of the page table and physical memory size. In our tests above, we vary the values from 16 MB to 85 MB and note that a 2-level page table would likely reduce the page table memory requirements significantly.

The ASM lookup rates are bound almost entirely by the cost of the address translation instructions, thus showing that our address translator is not bandwidth bound. The total frame rate is dominated ($\sim$85%) by the cost of the $O(n \log n)$ stream compaction algorithm [Hillis and Steele Jr. 1986; Horn 2005] portion of the refinement algorithm. This computation greatly reduces CPU read back cost at the expense of GPU computation (a net win), but a more efficient algorithm for this operation would further improve our frame rates. An additional bottleneck is the readback and CPU traversal of the compacted image. The size of this shadow page request image varies from tens to tens-of-thousands of pixels and is especially large when the light position changes. 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.

6.5 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 6.3 that supports quadlinear (mipmap) filtering. We demonstrate interactive painting of an 817k polygon model with effective paint resolutions varying between 64^3 to 2048^3 (see Figure 8).

Interactive painting of complex or unparameterized 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 atlases 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 made authoring octree textures very difficult. Lefebvre et al. [2004], in parallel with our work, implemented a GPU-based octree-like structure (see Table II). In contrast, our structure supports uniform, $O(1)$ accesses and supports quadlinear filtering without impacting application frame rates.

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. A 3D physical memory format enables the GPU to perform native trilinear filtering. We use the normalized coordinates of the rest pose of the model as texture coordinates.

The GPU use of our painting structure (without mipmapping) was completely defined within the core Glift functionality without application-specific modifications. As with the ASM structure, using the octree structure in a Cg shader is similar to a conventional 3D texture access:

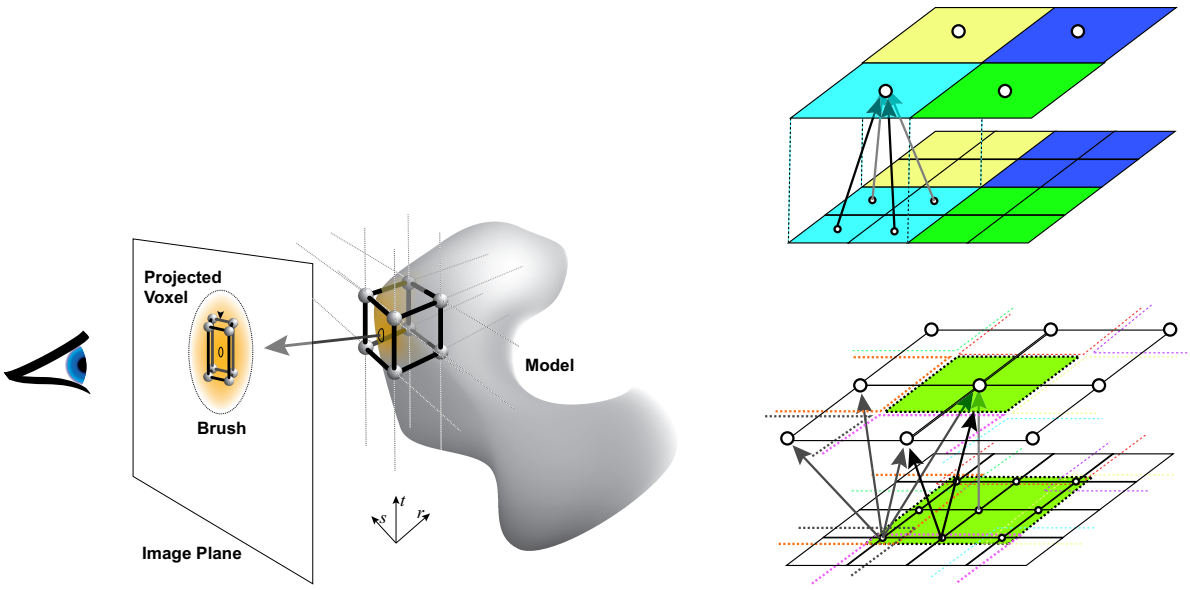
```
float4 main( uniform VMem3D octreePaint, float3 objCoord  ) : COLOR {
   return octreePaint.vTex3D( objCoord );
}
```

With the data structure already defined in Glift, the bulk of development time for the application was spent implementing brushing techniques, proper mipmap filtering, and adaptive refinement. The process of painting into the octree involves several steps: texture coordinate identification, brush rasterization, page allocation, texel update, and mipmap filtering. The first step, texture coordinate identification, uses the GPU to rasterize the model's texture coordinates for locating brush-model intersections.

We implement efficient and accurate brushes by using 2D brush profiles. The approach is motivated by the fact that octree voxel centers are rarely located directly on the surface being painted. This makes it difficult to create smooth 3D brush profiles without artifacts. To accommodate this, we project each texel into screen space and update the color using 2D brush profiles (Figure 9(a)). Our system restricts the minimum brush size to be no smaller than one screen space pixel, although this may translate into a larger brush in model space. We use the model-space brush size to automatically determine the resolution of the octree page. With this approach, larger brushes apply paint to coarser texture resolutions than would a smaller brush.

Since octree textures are sparse data structures, before a texel can be written we must ensure that it has been allocated at the desired resolution. When the desired brush resolution is finer than the existing texture resolution, the brick is re-allocated at the finer resolution and the previous texture color is interpolated into the higher resolution brick (i.e., texture refinement). Recall that because we are leveraging native GPU trilinear filtering, texels on the border of a texture brick are replicated by the neighboring bricks. This necessitates that we allocate neighboring tiles and write to all replicated texels when updating a brick-border texel. The common and tedious problem of allocation and writing to shared texels was folded into a specialized subclass of the more general Glift sparse, N-D, paged data structure, which greatly simplified the application-specific brushing algorithms. Since brushes always cover more than one texel and require blending with the previous texture color, we found it important to both cache the octree texture in CPU memory (eliminating unnecessary texture readback) and create a copy of the current brush at the appropriate brick resolution before blending it with the previous texture colors. Glift's ability to easily map sub-regions (individual pages) of the octree to CPU memory greatly simplified brushing updates.

Once a brush stroke is completed, we must update the mipmap hierarchy levels of the octree texture. In our implementation, we unify the notion of mipmap level and brick resolution level. This is implemented within the Glift framework by mipmapping only the virtual page table and sharing a single physical memory buffer between all mipmap levels. When the mipmap level is finer than or equal to the resolution of the painted texture, they utilize the same physical brick. However, when the mipmap level is coarser than the painted texture resolution, we must down sample the texture. Down sampling texels in our octree structure is a non-trivial issue due to the fact that our representation is node-centered as opposed to the cell-centered representation used by ordinary OpenGL textures (Figure 9(b)). With the node-centered scheme, some finer mipmap level texels are covered by the foot print of multiple coarser mipmap level texels. This requires the filtering to use a weighted average, with weights inversely proportional to the number of coarse mipmap level texels that share a specific finer mipmap level texel.



(a) Our paint system uses smooth 2D brush profiles to paint into 3D texels. We intersect the model and brush by using the screen-space projection of the 3D texture coordinates. The texture coordinates are the normalized rest pose of the model.

(b) Our octree paint structure supports quadlinear mipmapping. Mipmap generation requires special care because of the node-centered representation. This figure shows cell-centered (top) versus node-centered (bottom) filtering. Note the difference in filter support: cell-centered has a 2×2 support, whereas node-centered has 3×3 with some fine nodes shared by multiple coarse nodes.

Fig. 9: Brush profile and filtering detail for our painting system.

Because we are painting 2D surfaces using 3D bricks, there may be many texels in a brick that are never painted, since they do not intersect the surface. It is important that these texels do not contribute to the mipmap filtering process. We mark texels as having been painted by assigning them non-zero alpha values, having initialized the texture with zero alpha values. We set the alpha value to the resolution level at which the texel was last painted. This is done to ensure correct filtering. This subtle issue arises when combining texture refinement (interpolating coarse bricks into finer ones during the painting process) and mipmap filtering. When coarser texels are refined/interpolated into regions of a finer brick that do not intersect the surface, we end up with texels that cannot be updated by the finer brush, thus potentially resulting in erroneous mipmap filtering. In the mipmap filtering process, texels painted at finer resolutions are weighted higher than those that may have been interpolated from a lower resolution. While this approach does not completely fix the problem, since non-surface intersecting interpolated texels still have some contribution in the mipmap filtering, it tends to work well in practice.

The frame rates 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 900k polygons. The octree texturing operation did not affect frame rates in any of the viewing scenarios we tested. Frame rates 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 measured the performance impact of our data structure using synthetic tests similar to those shown in Figure 10. These synthetic results are invariant to changes in page sizes between 8^3 and 32^3. As such, lookups into our structure

Method	Cg ops	HW ops
Stream 1D→2D		
Glift, no specialization	8	5
Glift, with specialization	5	4
Brook (for reference)	—	4
Handcoded Cg	4	3
1D sparse, uniform 3D→3D page table		
Glift, no specialization	11	8
Glift, with specialization	7	5
Handcoded Cg	6	5
Adaptive shadow map + offset		
Glift, no specialization	31	10
Glift, with specialization	27	10
Handcoded Cg	16	9

Table III: 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.

are bound by the address translation instructions rather than the memory accesses (see Section 8).

7. RESULTS

7.1 Static Analysis of Glift GPU Code

GPU 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 instruction count 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 7.1 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 6.4 combined with an additional call to compute the page offset. This last case is a particularly difficult one for optimization because it contains a numerous redundant instructions executed by multiple method calls on the same structure.

All results in this section were computed on a 2.8 GHz Pentium 4 AGP 8x system with 1 GB of RAM and an NVIDIA GeForce 6800 GT with 256 MB of RAM, running Windows XP. The NVIDIA driver version was 75.80 and the Cg compiler is version 1.4 beta.

The results for these address translators and others we have tested show that the performance gap between programming with Glift and handcoding is minimal if any. The careful use of partial template specialization, Cg program specialization, and improved optimizations of recent compilers and drivers make it possible to express abstract structures with very little or no performance penalty.

7.1.1 *Memory Access Coherency.* Like their CPU counterparts, many GPU data structures use one or more levels of memory indirection to implement sparse or adaptive structures. This section assesses the cost of indirect GPU memory accesses and the relative importance of coherency in those accesses. Examples include page-table structures such the adaptive data structure described in Section 6.3.1 and tree structures [Foley and Sugerman 2005; Lefebvre et al. 2004; Thrane and Simonsen 2005].

This evaluation provides guidance in choosing the page size and number of levels of indirection to use in an address translator. 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 10). The experiment simulates memory accesses performed by paged structures by performing coherent accesses with page-sized regions, but randomly distributing the pages in physical memory. As such, the memory accesses are completely random when the page size is one (single pixel). Accesses are perfectly coherent when the page size is the entire viewport. To validate our experimental setup,

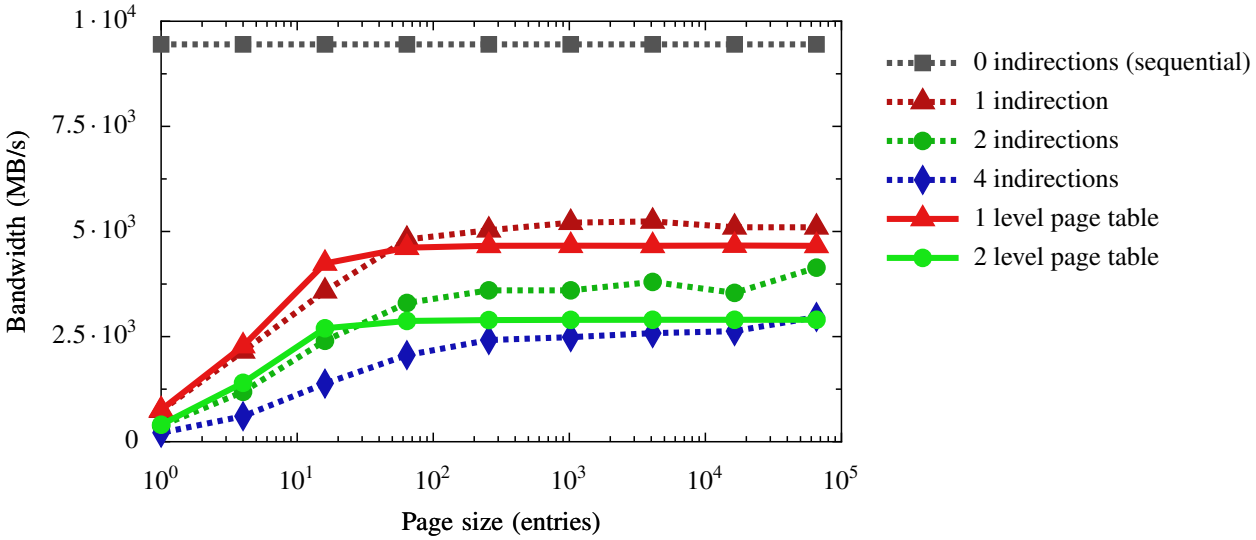


Fig. 10: Bandwidth as a function of page size for n-level chained indirect lookups (using no address computation) and for n-level page tables using our framework. Bandwidth figures only measure the data rate of the final lookup into physical memory and not the intermediate memory references. All textures are RGBA8 format. The results indicate that, for an NVIDIA GeForce 6800 GT, peak bandwidth can be achieved with pages larger than 8×8 for small levels of indirection and 16×16 for larger levels of indirection.

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. 2004].

We can draw several conclusions from Figure 10. First, the performance of a n-level page table is only slightly less than the performance of a n-level indirect lookup, indicating that our address translation code does not impose a significant performance penalty. The slightly better performance of the page table translator for larger page sizes is likely due to the page table being smaller than the indirection test's address texture. The page table size scales inversely to the page size, whereas the indirection test uses a constant-sized address texture. The page table translators are bandwidth bound when the pages contains less than 64 entries and are bound by arithmetic operations beyond that. Lastly, higher levels of indirection require larger page sizes to maintain peak GPU bandwidth. The results show that pages must contain at least 64 entries to maximize GPU bandwidth for a small number of indirections and contain 256 entries for higher levels of indirection.

These results show that it is not necessary to optimize the relative placement of physical pages in a paged data structure if the pages are larger than a minimum size (16×16 4-byte elements for the NVIDIA GeForce 6800 GT). This validates the design decision made in Section 6.3.1 not to solve the NP-hard *bin packing* problem when allocating ND pages in an ND buffer.

8. DISCUSSION

8.1 Language Design

Results from high-performance computing [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 have shown how to add template-like support to an existing GPU language with minimal changes to the shader compilation pipeline. We strongly encourage GPU language designers to add full support for static polymorphism and program specialization to their languages.

8.2 Separation of Address Translation and Physical Application Memory

In our implementation, address translation memory is separate from physical application memory. This separation has several advantages:

—**Multiple physical buffers mapped by one address translator** Mapping data to multiple physical memory buffers with a single address translator lets Glift store an "unlimited" amount of data in data structure nodes.
—**Multiple address translators for one physical memory buffer** Using multiple address translators for a single physical memory buffer enables Glift to reinterpret data in-place. For example, Glift can "cast" a 4D array into a 1D stream by simply changing the address translator.
—**Efficient GPU writes** Separating address translator and physical data allows the data in each to be laid out contiguously, thus enabling efficient GPU reading and writing. The separation also clarifies if the structure is being altered (write to address translator) or if only the data is being modified (write to physical memory).
—**Future GPU architecture optimizations** In CPUs, access to application memory is optimized through data caches, while access to address translation memory uses the specialized, higher-performance translation lookaside buffer (TLB). Future GPU hardware may optimize different classes of memory accesses in a similar way.

8.3 Iterators

The Glift iterator abstraction clarifies the GPU computation model, encapsulates an important class of GPGPU optimizations, and provides insight to ways that future hardware can better support computation.

8.3.1 *Generalization of GPU Computation Model.* Glift's iterator abstraction generalizes the stream computation model and clarifies the class of data structures that can be supported by current and future GPU computation. The stream model, popularized by Brook, describes computation of a kernel over elements of a stream. Brook defines a stream as a 1D–4D collection of records. In contrast, Glift defines GPU computation in terms of parallel iteration over a range of an arbitrary data structure. The structure may be as simple as an array or as complex as an octree. If the structure supports parallel iteration over its elements, it can be processed by the GPU. Given that the stream model is a subset of the Glift iterator model, GPU stream programming environments can use Glift as their underlying memory model to support more complex structures than arrays.

8.3.2 *Encapsulation of Optimizations.* Glift iterators capture an important class of optimizations for GPGPU applications. Many GPGPU papers describe elaborate computation strategies for pre-computing memory addresses and specifying computation domains by using a mix of CPU, vertex shader, and rasterizer calculations. Examples include pre-computing all physical addresses, drawing many small quads, sparse computation using stencil or depth culling, and Brook's iterator streams [Bolz et al. 2003; Buck et al. 2004; Govindaraju et al. 2005; Lefohn et al. 2004; Strzodka and Telea 2004]. In most cases, these techniques are encapsulated by Glift iterators and can therefore be separated from the algorithm description. An exciting avenue of future work is generating GPU iterators entirely on the GPU using render-to-vertex-array, vertex texturing, and future architectural features for generating geometry [Beyond3D 2003].

8.3.3 *Explicit Memory Access Patterns.* Glift's current element iterator implementation performs random-access reads for all iterator access patterns (single, neighborhood, and random) because current hardware does not support this distinction. However, the illusion to the programmer remains one of limited stream or neighbor access. While a current GPU may not be able to benefit from this abstraction, current CPUs, multiple-GPU configurations, future GPUs, and other parallel architectures may be able to schedule memory loads much more efficiently for programs written with these explicit access pattern constructs.

8.3.4 *Parallel Iteration Hardware.* Future GPUs could offer better support for computation over complex data structures by providing true pointers and native support for incrementing those pointers in parallel. Glift's current GPU iterators give the illusion of pointers by combining textures and address iterators. This approach can require multiple address translation steps to map between the output domain, a data structure's virtual domain, and a structure's physical domain. It is possible that future architectures will optimize this by providing native support for parallel range iterators. Instead of generating computation with a rasterizer, computation would be generated via parallel incrementing of iterators.

8.4 Near-Term GPU Architectural Changes

We note how near-term suggested improvements in graphics hardware would impact the performance and flexibility of Glift:

—**Integers**: Hardware support for integer data types would both simplify address translation code and avoid the precision difficulties with floating-point addressing [Buck 2005].
—**Vertex Texture Performance**: Improvements in the performance of vertex textures will simplify and optimize Glift iterator implementations.
—**Per-Fragment Mipmapping**: In current NVIDIA GPUs, accessing mipmap textures with the explicit level-of-detail (LOD) instruction (TXL) results in all four fragments in a 2×2 pixel block accessing the same mipmap level, essentially ignoring three of the four LOD values. Workarounds for this behavior are error-prone and computationally expensive, limiting the utility of mipmaps as general data structure primitives[4].

8.5 Limitations

The address translation abstraction we have presented in this paper does not currently represent all possible data structures. In particular, we have not investigated GPU implementations of non-indexable structures such as linked lists, graphs, and mesh connectivity structures. 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.

The Glift implementation is a proof-of-concept with significant room for improvement. For example, Glift does not currently virtualize the total amount of GPU memory, virtualize memory on multiple GPUs, or provide file formats for its structures (although the textures can be saved using any compatible image format). In addition, the CPU-side of Glift does not yet take advantage of SIMD instructions or support multithreading.

9. CONCLUSIONS

This paper demonstrates an efficient, high-level abstraction for GPU random-access data structures for both graphics and GPGPU applications. The abstraction factors GPU data structures into four components: physical memory, programmable address translation, virtual memory, and iterators. We implement the abstraction as a C++ and Cg template library called Glift and show that the resulting code is nearly as efficient as low-level, hand-written code.

Glift makes it possible to separate GPU data structure and algorithm development and description. We demonstrate how this separation reduces programming complexity in multiple ways. First, we present simple examples which demonstrate the reduction in code complexity and clear separation of data structures and algorithm. Second, we characterize a large number of previous GPU data structures in terms of Glift abstractions, thereby illuminating many similarities between seemingly diverse structures. Third, we describe novel, complex GPU data structures—a GPU stack, quadtree, and octree—in terms of generic Glift components. Lastly, we demonstrate two applications of these

[4]During the final stages of preparation of this paper, we were alerted to the fact that the explicit derivative texture instruction (TXD) does perform correct per-fragment mipmap lookups. The results in this paper do not reflect this insight.

structures, neither previously demonstrated on GPUs: adaptive shadow maps and octree 3D paint. The implementation and description of these applications is greatly simplified by the Glift abstraction and the separation of algorithm from data structures.

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 the Standard Template Library (STL) and Boost have become integral in CPU program development, an efficient GPU data structure abstraction makes it possible for vendors to offer implementations optimized for their architecture and application developers to more easily create complex applications.

APPENDIX

This appendix presents the complete C++ source code for the 4D array example in Section 6.1. The purpose of this appendix is to show the source code before and after transforming it with Glift.

The C++ source for the non-Glift example is:

```
// ...  Initialize OpenGL rendering context ...
vec4i virtArraySize(10, 10, 10, 10); // Compute sizes

int numElts = 0;
for (int i = 0; i < virtArraySize.size(); ++i) {
  numElts += virtArraySize[i];
}
vec2i physArraySize = int( ceilf( sqrtf( numElts ) ) );

vec4f sizeConst(1, 1, 1, 1);
sizeConst[1] = virtArraySize[0];
sizeConst[2] = virtArraySize[0] * virtArraySize[1];
sizeConst[3] = virtArraySize[0] * virtArraySize[1] * virtArraySize[2];

// Allocate 2 arrays that hold vec4f values
GLuint array1_texId;
glGenTextures(1, &array1_texId);
glBindTexture(GL_TEXTURE_2D, array1_texId);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F_ARB,
             physArraySize.x(), physArraySize.y(),
             GL_RGBA, GL_FLOAT, NULL);

GLuint array2_texId; glGenTextures(1, &array2_texId);
glBindTexture(GL_TEXTURE_2D, array1_texId);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F_ARB,
             physArraySize.x(), physArraySize.y(),
             GL_RGBA, GL_FLOAT, NULL);

// Create Cg shader
CGprogram prog = cgCreateProgramFromFile( cgCreateContext(), CG_SOURCE, "laplacian.cg",
                                          CG_PROFILE_FP40, "main", NULL );

// Bind shader and enable programmable fragment pipeline
cgEnableProfile(CG_PROFILE_FP40);
cgGLBindProgram(prog);

// Bind parameters to shader
CGparameter array1Param = cgGetNamedParameter(prog, "array1");
cgGLSetTextureParameter(array1Param, array1_texId);
cgGLEnableTextureParameter(array1Param);
```

```
CGparameter virtSizeParam = cgGetNamedParameter(prog, "virtSize");
cgSetParameter4v( virtSizeParam, virtArraySize.data() );

CGparameter physSizeParam = cgGetNamedParameter(prog, "physSize");
cgSetParameter2v( physSizeParam, physArraySize.data() );

CGparameter sizeConstParam = cgGetNamedParameter(prog, "sizeConst");
cgSetParameter4v( sizeConstParam, sizeConst.data() );

// Specialize size parameters
cgSetParameterVariability(virtSizeParam,  CG_LITERAL);
cgSetParameterVariability(physSizeParam,  CG_LITERAL);
cgSetParameterVariability(sizeConstParam, CG_LITERAL);

// Compile and load shader
cgCompileProgram(prog);
cgGLLoadProgram(prog);

// Create Framebuffer Object and attach "array2" to COLOR0
GLuint fboId;
glGenFramebuffersEXT(1, &fboId);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, array2_texId, 0 );

// Render screen-aligned quad (physArraySize.x(), physArraySize.y())
glMatrixMode(GL_PROJECTION); glLoadIdentity();
glMatrixMode(GL_MODELVIEW); glLoadIdentity();
glViewport(0, 0, physArraySize.x(), physArraySize.y());
glBegin(GL_QUADS);
   glVertex2f(-1.0f, -1.0f); glVertex2f(+1.0f, -1.0f);
   glVertex2f(+1.0f, +1.0f); glVertex2f(-1.0f, +1.0f);
glEnd();
```

In contrast, the C++ code for the Glift version is:

```
// ...  Initialize OpenGL rendering context ...
typedef glift::ArrayGpu<vec4i, vec4f> ArrayType;

// Allocate 2 arrays that hold vec4f values
vec4i virtArraySize(10, 10, 10, 10);
ArrayType array1( virtArraySize );
ArrayType array2( virtArraySize );

// Create Cg shader
CGprogram prog = cgCreateProgramFromFile( cgCreateContext(), CG_SOURCE, "laplacianGlift.cg",
                                          CG_PROFILE_FP40, "main", NULL);

// Instantiate Glift Cg types
GliftType arrayTypeCg = glift::cgGetTemplateType<ArrayType>();
prog = glift::cgInstantiateParameter( prog, "array1", arrayTypeCg );

GliftType iterTypeCg = glift::cgGetTemplateType<ArrayType::gpu_iterator>();
prog = glift::cgInstantiateParameter( prog, "it", iterTypeCg );

// Get GPU range iterator for all elements in array2
// - This is a neighborhood iterator that permits relative
// indexing within the specified neighborhood.
vec4i origin(0, 0, 0, 0);
vec4i size = virtArraySize;
vec4i minNeighborhood(-1, -1, -1, -1);
vec4i maxNeighborhood(+1, +1, +1, +1);
ArrayType::gpu_neighbor_range rit = array2.gpu_neighbor_range( origin, size, minNeighborhood, maxNeighborhood );

// Bind parameters to shader
CGparameter array1Param = cgGetNamedParameter(prog, "array1");
array1.bind_for_read( array1Param );

CGparameter iterParam = cgGetNamedParameter(prog, "it");
rit.bind_for_read( iterParam );
```

```
// Specialize size parameters
array1.set_member_variability( array1Param, CG_LITERAL );
rit.set_member_variability( iterParam, CG_LITERAL );

// Compile and load program
cgCompileProgram(prog);
cgGLLoadProgram(prog);

// Bind shader and enable programmable fragment pipeline
cgEnableProfile(CG_PROFILE_FP40);
cgGLBindProgram(prog);

// Create Framebuffer Object and attach "array2" to COLOR0
GLuint fboId;
glGenFramebuffersEXT(1, &fboId);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);

array2.bind_for_write( fboId, GL_COLOR_ATTACHMENT0_EXT );

// Compute across domain specified by address iterator
glift::exec_gpu_iterators( rit );
```

Note that the Glift version for this simple example is approximately half the number of lines of code. The savings is significantly more for complex structures containing multiple address translation and data textures. The intent of the Glift version is much clearer than the raw OpenGL/Cg version, yet Glift remains a low-enough level library that it can be easily integrated into existing C++/OpenGL/Cg programming environments.

ACKNOWLEDGMENTS

We would like to thank Fabio Pellacini for his invaluable input on the adaptive shadow and octree painting applications. In addition, Ross Whitaker, Milan Ikits, and Mike Houston provided early feedback on the adaptive data structure idea; and Daniel Horn helped with the stream compaction algorithm. We also thank the following people for providing feedback on the work throughout its development: Jim Ahrens, David Blythe, Ian Buck, Randima Fernando, Dominik Göddeke, Mark Harris, Adam Moerschell, Pat McCormick, Matt Papakipos, Matt Pharr, Mark Segal, Peter-Pike Sloan, Dan Wexler, and the anonymous reviewers. We also thank NVIDIA for donating graphics hardware and software support time to this project, specifically NVIDIA engineers Craig Kolb, Nick Triantos, and Cass Everitt, who provided substantial Cg and display driver support. Yong Kil, Chris Co, and Fabio Pellacini provided the 3D models. We lastly would like to thank the funding agencies that made it possible to pursue this work: National Science Foundation Graduate Fellowship (Aaron Lefohn), Department of Energy High-Performance Computer Science Graduate Fellowship (Joe Kniss), Department of Energy VIEWs program, Department of Energy Early Career Principal Investigator Award #DE-FG02-04ER25609, Los Alamos National Laboratory, Chevron, UC MICRO grant #04-070, and the Caesar Research Institute.

REFERENCES

ALEXANDRESCU, A. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley.

AN, P., JULA, A., RUS, S., SAUNDERS, S., SMITH, T., TANASE, G., THOMAS, N., AMATO, N., AND RAUCHWERGER, L. 2001. STAPL: An adaptive, generic parallel C++ library. In *Workshop on Languages and Compilers for Parallel Computing*. 193–208.

AUSTERN, M. H., TOWLE, R. A., AND STEPANOV, A. A. 1996. Range partition adaptors: A mechanism for parallelizing STL. *ACM SIGAPP Applied Computing Review 4,* 1, 5–6.

BENSON, D. AND DAVIS, J. 2002. Octree textures. *ACM Transactions on Graphics 21,* 3 (July), 785–790.

BEYOND3D. 2003. DirectX next early preview. `http://www.beyond3d.com/articles/directxnext/`.

BINOTTO, A. P. D., COMBA, J. L. D., AND FREITAS, C. M. D. 2003. Real-time volume rendering of time-varying data using a fragment-shader compression approach. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. 69–75.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics 22,* 3 (July), 917–924.

BOOST. 2005. Boost C++ libraries. http://www.boost.org/.

BUCK, I. 2005. Taking the plunge into GPU computing. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Chapter 32, 509–519.

BUCK, I., FATAHALIAN, K., AND HANRAHAN, P. 2004. GPUBench: Evaluating GPU performance for numerical and scientific applications. In *2004 ACM Workshop on General-Purpose Computing on Graphics Processors*. C–20.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics 23,* 3 (Aug.), 777–786.

CARR, N. A. AND HART, J. C. 2004. Painting detail. *ACM Transactions on Graphics 23,* 3 (Aug.), 845–852.

CHRISTENSEN, P. H. AND BATALI, D. 2004. An irradiance atlas for global illumination in complex production scenes. In *Rendering Techniques 2004*. 133–141.

COLE, M. AND PARKER, S. 2003. Dynamic compilation of C++ template code. In *Scientific Programming*. Vol. 11. IOS Press, 321–327.

COOMBE, G., HARRIS, M. J., AND LASTRA, A. 2004. Radiosity on graphics hardware. In *Proceedings of the 2004 Conference on Graphics Interface*. 161–168.

DEBRY, D., GIBBS, J., PETTY, D. D., AND ROBINS, N. 2002. Painting and rendering textures on unparameterized models. *ACM Transactions on Graphics 21,* 3 (July), 763–768.

ERNST, M., VOGELGSANG, C., AND GREINER, G. 2004. Stack implementation on programmable graphics hardware. In *Proceedings of Vision, Modeling, and Visualization*. 255–262.

FERNANDO, R., FERNANDEZ, S., BALA, K., AND GREENBERG, D. P. 2001. Adaptive shadow maps. In *Proceedings of ACM SIGGRAPH 2001*. Computer Graphics Proceedings, Annual Conference Series. 387–390.

FOLEY, T. AND SUGERMAN, J. 2005. KD-Tree acceleration structures for a GPU raytracer. In *Graphics Hardware 2005*. 15–22.

GOODNIGHT, N., WOOLLEY, C., LEWIN, G., LUEBKE, D., AND HUMPHREYS, G. 2003. A multigrid solver for boundary value problems using programmable graphics hardware. In *Graphics Hardware 2003*. 102–111.

GOVINDARAJU, N. K., RAGHUVANSHI, N., HENSON, M., TUFT, D., AND MANOCHA, D. 2005. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Tech. Rep. TR05-016, University of North Carolina.

GUENTER, B., KNOBLOCK, T. B., AND RUF, E. 1995. Specializing shaders. In *Proceedings of SIGGRAPH 95*. Computer Graphics Proceedings, Annual Conference Series. 343–350.

HARRIS, M. AND BUCK, I. 2005. GPU flow control idioms. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Chapter 34, 547–555.

HARRIS, M. J., BAXTER III, W., SCHEUERMANN, T., AND LASTRA, A. 2003. Simulation of cloud dynamics on graphics hardware. In *Graphics Hardware 2003*. 92–101.

HILLIS, W. D. AND STEELE JR., G. L. 1986. Data parallel algorithms. *Communications of the ACM 29,* 12 (Dec.), 1170–1183.

HOEL, E. G. AND SAMET, H. 1995. Data-parallel primitives for spatial operations. In *Proceedings of the 1995 International Conference on Parallel Processing*. III:184–191.

HORN, D. 2005. Stream reduction operations for GPGPU applications. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Chapter 36, 573–589.

JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular Z-buffer: Hardware acceleration for irregular data structures. *ACM Transactions on Graphics 24,* 4 (Oct.), 1462–1482.

KARMESIN, S., HANEY, S., HUMPHREY, B., CUMMINGS, J., WILLIAMS, T., CROTINGER, J., SMITH, S., AND GAVRILOV, E. 2002. Pooma: Parallel object-oriented methods and applications. http://acts.nersc.gov/pooma/.

KENDALL, R. A., SOSONKINA, M., GROPP, W. D., NUMRICH, R. W., AND STERLING, T. 2005. Parallel programming models applicable to cluster computing and beyond. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, A. M. Bruaset and A. Tveito, Eds. Lecture Notes in Computational Science and Engineering, vol. 51. Springer-Verlag.

KESSENICH, J., BALDWIN, D., AND ROST, R. 2004. The OpenGL Shading Language version 1.10.59. http://www.opengl.org/documentation/oglsl.html.

KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., AND SUMNER, F. H. 1962. One-level storage system. *IRE Transactions on Electronic Computers EC-11*, 223–235.

KRAUS, M. AND ERTL, T. 2002. Adaptive texture maps. In *Graphics Hardware 2002*. 7–16.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2004. All-purpose texture sprites. Tech. Rep. 5209, INRIA. May.

LEFOHN, A., KNISS, J., AND OWENS, J. 2005. Implementing efficient parallel data structures on GPUs. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Chapter 33, 521–545.

LEFOHN, A. E., KNISS, J. M., HANSEN, C. D., AND WHITAKER, R. T. 2003. Interactive deformation and visualization of level set surfaces using graphics hardware. In *IEEE Visualization 2003*. 75–82.

LEFOHN, A. E., KNISS, J. M., HANSEN, C. D., AND WHITAKER, R. T. 2004. A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces. *IEEE Transactions on Visualization and Computer Graphics 10,* 4 (July/Aug.), 422–433.

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

LOSASSO, F., GIBOU, F., AND FEDKIW, R. 2004. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics 23,* 3 (Aug.), 457–462.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics 22,* 3 (July), 896–907.

MCCOOL, M., TOIT, S. D., POPA, T., CHAN, B., AND MOULE, K. 2004. Shader algebra. *ACM Transactions on Graphics 23,* 3 (Aug.), 787–795.

MCCORMICK, P. S., INMAN, J., AHRENS, J. P., HANSEN, C., AND ROTH, G. 2004. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *IEEE Visualization 2004*. 171–178.

NVIDIA DEVELOPER RELATIONS. 2003. Cg: C for graphics. `http://developer.nvidia.com/`.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E., AND PURCELL, T. 2005. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*. 21–51.

PHARR, M. 2004. An introduction to shader interfaces. In *GPU Gems*, R. Fernando, Ed. Addison Wesley, Chapter 32, 537–550.

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

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21,* 3 (July), 703–712.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *Graphics Hardware 2003*. 41–50.

SCHNEIDER, J. AND WESTERMANN, R. 2003. Compression domain volume rendering. In *IEEE Visualization 2003*. 293–300.

SEN, P. 2004. Silhouette maps for improved texture magnification. In *Graphics Hardware 2004*. 65–74.

SEN, P., CAMMARANO, M., AND HANRAHAN, P. 2003. Shadow silhouette maps. *ACM Transactions on Graphics 22,* 3 (July), 521–526.

STAMMINGER, M. AND DRETTAKIS, G. 2002. Perspective shadow maps. *ACM Transactions on Graphics 21,* 3 (July), 557–562.

STRZODKA, R. AND TELEA, A. 2004. Generalized distance transforms and skeletons in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization (VisSym '04)*. 221–230.

TARINI, M., HORMANN, K., CIGNONI, P., AND MONTANI, C. 2004. PolyCube-Maps. *ACM Transactions on Graphics 23,* 3 (Aug.), 853–860.

THRANE, N. AND SIMONSEN, L. O. 2005. A comparison of acceleration structures for GPU assisted ray tracing. M.S. thesis, University of Aarhus.

WIMMER, M., SCHERZER, D., AND PURGATHOFER, W. 2004. Light space perspective shadow maps. In *Eurographics Symposium on Rendering*. 143–151.

Received January 2005; revised October 2005; accepted December 2005.