Glift: Generic, Efficient Random-Access GPU Data Structures

Aaron Lefohn
University of California, Davis
Problem Statement

• **Goal**
  • Simplify creation and use of random-access GPU data structures for graphics and GPGPU programming

• **Contributions**
  • Abstraction for GPU data structures
  • Glift template library
  • Iterator computation model for GPUs
Collaborators

- Joe Kniss
  University of Utah

- Robert Strzodka
  Stanford University

- Shubhabrata Sengupta
  University of California, Davis

- John Owens
  University of California, Davis
Many Interesting GPU Data Structures

- Photon map
- Sparse matrix
- Sparse simulation grid
- Polycube (3D grid, cubeMap, ...)
- N-tree

- But...
  - No way to distribute/reuse implementations
  - Complexity stifles innovation

Motivation
CPU Software Development

**Motivation**

Application

- Data Structure Library
- Algorithm Library
- CPU Memory

**Benefits**
- Algorithms and data structures expressed in problem domain
- Decouple algorithms and data structures
- Code reuse
GPU Software Development

Motivation

Application
- Data structure and algorithm

GPU Memory

• **Problems**
  - Code is tangled mess of algorithm and data structure access
  - Algorithms expressed in GPU memory domain
  - No code reuse
GPU Data Structures

• What’s Missing?
  • Standalone abstraction for GPU data structures for graphics or GPGPU programming
Simple Example

- **CPU (C++)**

```c
float srcData[10][10][10];
float dstData[10][10][10];

... initialize data ...

for (size_t z = 1; z < 10; ++z) {
    for (size_t y = 1; z < 10; ++y) {
        for (size_t x = 1; z < 10; ++x) {
            dst[z][y][x] = log( 1 + src[z][y][x] );
        }
    }
}
```
We Want To Transform This...

- GPU (Cg)

```c
float3 getAddr3D( float2 winPos, float2 winSize, float3 sizeConst3D ) {
    float3 curAddr3D;
    float2 winPosInt = floor(winPos);
    float addr1D = winPosInt.y * winSize.x + winPosInt.x;
    addr3D.z  = floor( addr1D / sizeConst3D.z );
    addr1D   = addr3D.z * sizeConst3D.z;
    addr3D.y  = floor( addr1D / sizeConst3D.y );
    addr3D.x  = addr1D - addr3D.y * sizeConst3D.y;
    return addr3D;
}

float3 logAlg(uniform samplerRECT data,
               uniform float2 winSize,
               uniform float3 sizeConst3D,
               float2 winPos : WPOS ) : COLOR
{
    float3 addr3D = getAddr3D( winPos, winSize, sizeConst3D );
    float data   = texRECT(data, addr3D );
    return log( 1 + data );
}
```
We Want To Transform This...

• GPU (Cg and C++)

```cpp
float3 getAddr3D( float2 winPos, float2 winSize, float3 sizeConst3D ) {
    float3 curAddr3D;
    float2 winPosInt = floor(winPos);
    float addr1D = winPosInt.y * winSize.x + winPosInt.x;
    addr1D -= floor( addr1D / sizeConst3D.z );
    addr3D.y = floor( addr1D / sizeConst3D.y );
    addr3D.x = addr1D - addr3D.y * sizeConst3D.y;
    return addr3D;
}

float3 logAlg(uniform samplerRECT data, uniform float2 winSize, uniform float3 sizeConst3D, float2 winPos : WPOS ) : COLOR {
    float3 addr3D = getAddr3D( winPos, winSize, sizeConst3D );
    float data = texRECT(data, addr3D );
    return log( 1 + data );
}
```

Motivation

GLuint srcDataId = 1;
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, srcDataId);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_WRAP_T, GL_CLAMP);
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_LUMINANCE32F_ARB, 0, 0, 40, 40, GL_LUMINANCE, NULL);

GLuint dstDataId = 2;
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, dstDataId);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_WRAP_T, GL_CLAMP);
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_LUMINANCE32F_ARB, 0, 0, 40, 40, GL_LUMINANCE, NULL);

... Initialize data ...
```
Into This.

• GPU (C++ and Cg with Glift)

typedef glift::ArrayGpu<vec3i, vec1f> ArrayType;
ArrayType src(vec3i(10, 10, 10));
ArrayType dst(vec3i(10, 10, 10));

... initialize data ...

float logAlg(ElementIter srcData) : COLOR
{
    return log(1 + srcData.value());
}
Overview

- Motivation and Previous Work
- Abstraction
- Implementation
- Examples
- Conclusions
Abstraction Design Goals

- **GPU data structure abstraction that**
  - Enables easy creation of new structures
  - Is minimal abstraction of GPU memory model
  - Separates data structures and algorithms
  - Encourages efficiency
Building the Abstraction

**Approach**

- Bottom-up, working towards STL-like syntax
- Identify common patterns in GPU papers and code
- Inspired by
  - STL, Boost, Brook, STAPL, Stepanov
What is the GPU Memory Model?

- **CPU interface**
  - `glTexImage`  
  - `glDeleteTextures`  
  - `glTexSubImage`  
  - `glGetTexSubImage*`  
  - `glCopyTexImageSubImage`  
  - `glBindTexture`  
  - `glFramebufferTexture`  

```
CPU interface
\n• glTexImage: malloc
• glDeleteTextures: free
• glTexSubImage: memcpy
• glGetTexSubImage*: memcpy
• glCopyTexImageSubImage: memcpy
• glBindTexture: read-only
• glFramebufferTexture: write-only
```

* Does not exist. Emulate with `glReadPixels`
What is the GPU Memory Model?

• GPU Interface (shown in Cg)

  • uniform samplerND
  • texND(tex, addr)
  • varying floatN stream
  • stream

  data structure
  random-access read
  stream parameter declaration
  stream read
GPU Data Structure Abstraction

• Factor GPU data structures into
  • Physical memory
  • Virtual memory
  • Address translator
  • Iterators
Physical Memory

- Native GPU textures
  - Choose based on algorithm efficiency requirements
  - 1D, 2D, 3D, Cube, Mip
    - Dimensionality
    - Read-only vs. read-write
    - Point-sample vs. filtering
    - Maximum size
Virtual Memory

- **Virtual N-D address space**
  - Choose based on problem space of algorithm
  - Defined by physical memory and address translator

**Abstraction**

Virtual representation of memory: 3D grid

- Translation
  - 3D native mem
- Translation
  - 2D slices
- Translation
  - Flat 3D texture
Address Translator

- Mapping between physical and virtual addrs

- Core of data structure
- Small amount of code defines *all* required CPU and GPU memory interfaces
Address Translator

• Core of data structure
  • Extension point for creating new structures
  • Must define
    
    \texttt{translate(...)}
    \texttt{translate\_range(...)}
Address Translator Classifications

- **Representation**
  - Analytic / Discrete

- **Memory Complexity**
  - $O(1)$, $O(\log N)$, $O(N)$, ...

- **Compute Complexity**
  - $O(1)$, $O(\log N)$, $O(N)$, ...

- **Compute Consistency**
  - Uniform vs. non-uniform

- **Total / Partial**
  - Complete vs. sparse

- **One-to-one / Many-to-one**
  - Uniform vs. adaptive

Abstraction
Data Structure Examples

• Brook streams (Buck et al. 2004)

Abstraction

1D Virtual → 2D Physical
Data Structure Examples

- **Brook streams**
  - Physical address: 2D
  - Virtual address: N-D
  - Address translator: ND-to-2D
    - Analytic
    - O(1) memory
    - O(1) compute
    - Uniform consistency
    - Total, uniform mapping

(Buck et al. 2004)
Data Structure Examples

- Dynamic sparse 3D grid (Lefohn et al. 2003)

Virtual Domain → Page Table → Physical Memory
Data Structure Examples

- **Dynamic sparse 3D grid** (Lefohn et al. 2003)
  - Physical address: 2D
  - Virtual address: 3D
  - Address translator: 3D page table
    - Discrete
    - $O(N)$ memory
    - $O(1)$ compute
    - Uniform consistency
    - Partial, uniform mapping
Data Structure Examples

- Photon Map (kNN-grid) (Purcell et al. 2003)

*Image from “Implementing Efficient Parallel Data Structures on GPUs,” Lefohn et al., GPU Gems II, ch. 33, 2005*
## Data Structure Examples

<table>
<thead>
<tr>
<th>Photon Map (kNN-grid)</th>
<th>(Purcell et al. 2003)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical address</td>
<td>2D</td>
</tr>
<tr>
<td>Virtual address</td>
<td>3D</td>
</tr>
<tr>
<td>Address translator</td>
<td>3D page table</td>
</tr>
<tr>
<td></td>
<td>- Variable sized phys pages</td>
</tr>
<tr>
<td></td>
<td>- “Grid of lists”</td>
</tr>
</tbody>
</table>

- Discrete
- O(N) memory
- O(L) compute
- Non-uniform consistency
- Partial, adaptive mapping
Glift Iterators

- We’ve so far only discussed data access
- What about data structure traversal?
Iterators

• Separate algorithms and data structures
  • Minimal interface between data and algorithm
  • Required for GPGPU use of data structure
  • Encapsulate GPGPU optimizations
Iterators

• Abstract data access and traversal

```cpp
DataStructureType::iterator it;
for (it = data.begin(); it != data.end(); ++it)
{
    *it = -(*it);
}
```
Glift Iterators

- **Address iterators**
  - Iterator value is N-D address
  - GPU interpolants

- **Element iterators**
  - Iterator value is data structure element
  - C/C++ pointer, STL iterator, streams
Element Iterator Concepts

- **Permission**
  - Read-only, write-only, read-write

- **Access region**
  - Single, neighborhood, random

- **Traversal**
  - Forward, backward, parallel range
Which Element Iterators?

- **Read-only, single access, range iterator**
  - GPU stream input

- **Read-only, random-access, range iterator**
  - GPU texture input

- **Write-only, single access, range iterator**
  - GPU render target
Example 1: “Before” and “After” Glift

- Transform GPU code with Glift
Simple Example

• 3D Array with 2D physical memory

CPU (C++)

```
float srcData[10][10][10];
float dstData[10][10][10];

... initialize data ...
```

```
for (size_t z = 1; z < 10; ++z) {
    for (size_t y = 1; z < 10; ++y) {
        for (size_t x = 1; z < 10; ++x) {
            dstData[z][y][x] = srcData[z-1][y-1][x-1];
        }
    }
}
```
Example 1: Shader w/out Glift

```
float3 physToVirt(float2 pa, float2 physSize, float3 virtSizes) {
    float3 va;
    float addr1D = pa.y * physSize.x + pa.x;
    va.z = floor(addr1D / virtSizes.z);
    addr1D -= va.z * sizeConst3D.z;
    va.y = floor(addr1D / virtSizes.y);
    va.x = addr1D - va.y * virtSizes.y;
    return va;
}

float2 virtToPhys(float3 va, float2 physSize, float3 virtSizes) {
    float addr1D = dot(va, virtSizes);
    float normAddr1D = addr1D / physSize.x;
    float2 pa = float2(frac(normAddr1D) * physSize.x, normAddr1D);
}

float3 main(uniform samplerRECT physMem, uniform float2 physSize, uniform float3 virtSizes, float2 pa : WPOS) : COLOR {
    float3 va = physToVirt(floor(pa), physSize, virtSizes);
    float3 neighborAddr = va - float3(1, 1, 1);
    return texRECT(data, virtToPhys(neighborAddr3D, physSize, virtSizes));
}
```

**Physical-to-Virtual Address Translation**

**Virtual-to-Physical Address Translation**

**Physical Memory Read**
Example 1: Glift Components

```cpp
float3 physToVirt(float2 pa, float2 physSize, float3 virtSizes) {
    float3 va;
    float addr1D = pa.y * physSize.x + pa.x;
    va.z = floor(addr1D / virtSizes.z);
    addr1D -= va.z * sizeConst3D.z;
    va.y = floor(addr1D / virtSizes.y);
    va.x = addr1D - va.y * virtSizes.y;
    return va;
}
```

```cpp
float2 virtToPhys(float3 va, float2 physSize, float3 virtSizes) {
    float addr1D = dot(va, virtSizes);
    float normAddr1D = addr1D / physSize.x;
    float2 pa = float2(frac(normAddr1D) * physSize.x, normAddr1D);
}
```

```cpp
float3 main(uniform samplerRECT physMem, uniform float2 physSize, uniform float3 virtSizes, float2 pa : WPOS) : COLOR {
    float3 va = physToVirt(floor(pa), physSize, virtSizes);
    float3 neighborAddr = va - float3(1, 1, 1);
    return texRECT(data, virtToPhys(neighborAddr), physSize, virtSizes);
}
```
Example 1: GPU Shader with Glift

Cg Usage

```cpp
float3 main( uniform VMem3D srcData,
            AddrIter3D iter ) : COLOR
{
    float3 va = iter.value();
    return srcData.vTex3D( va - float3(1,1,1) );
}
```
Example 1: Glift Data Structures

C++ Usage

```cpp
vec3i origin(0,0,0);
vec3i size(10,10,10);

typedef ArrayGpu<vec3i,vec1f> ArrayType;
ArrayType srcData( size );
ArrayType dstData( size );

... initialize dataPtr ...
srcData.write( origin, size, dataPtr );

typedef ArrayType::addr_trans AddrTransType;
AddrTransType::gpu_range it =
    dstData.addr_trans().gpu_range(origin, size);

it.bind_for_read( iterCgParam );
srcData.bind_for_read( srcCgParam );
dstData.bind_for_write( COLOR0, myFrameBufferObject );

exec_gpu_iterators( it );
```
Overview

- Motivation
- Abstraction
- Implementation
- Examples
- Conclusions
Glift Components

Application

Container Adaptors

VirtMem

PhysMem

AddrTrans

C++ / Cg / OpenGL
Glift Design Goals

- Efficiency
- Easy, incremental adoption
- Easily extensible
- CPU/GPU interoperability
Glift Design Goals

- **Efficiency**
  - Static polymorphism (C++ and Cg)
  - Cg program specialization
  - Cg compiler optimizations

- **Easy, incremental adoption**

- **Easily extensible**

- **CPU/GPU interoperability**
Glift Design Goals

- **Efficiency**
- **Easy, incremental adoption**
  - Integrate with Cg/OpenGL/C++
  - STL-like and texture-like interfaces
  - Use components alone or composites
- **Easily extensible**
- **CPU/GPU interoperability**
Glift Design Goals

- Efficiency
- Easy, incremental adoption
- Easily extensible
  - Create new structure by:
    - Change behavior of existing address translator
    - New address translator
    - New container adaptor
- CPU/GPU interoperability
Glift Design Goals

- Efficiency
- Easy, incremental adoption
- Easily extensible
- CPU/GPU interoperability
  - Unified C++/Cg code base
  - Map memory to CPU or GPU
  - CPU and GPU iterators
C++/Cg Integration

- Each component defines C++ and Cg code
  - C++ objects have Cg struct representation
  - Stringified Cg parameterized by C++ templates

- Cg “template” instantiation
  - Insert generated Glift source code into shader

```cpp
glift::cgGetTemplateType<MyDataStructType>();
glift::cgInstantiateParameter(...);
```

- All other compilation/loading/binding identical to standard shader
Cg Compilation Example

- **Cg code**

```cpp
float4 main( uniform VMem3D octree,
            float3 coord ) : COLOR {
    return octree.vMem3D(coord);
}
```

- **C++ code**

```cpp
typedef OctreeGPU<vec4ub> octree_type;
GliftType type = cgGetTemplateType<octree_type>();
CGprogram prog = cgCreateProgram(...);
prog = cgInstantiateParameter(prog, "octree", type);
cgCompileProgram(prog);
```
Overview

- Motivation and previous work
- Abstraction
- Case Study
  - Adaptive shadow maps and octree 3D paint
- Conclusions
Example 2: Adaptive Shadow Maps

- **Show Glift usage with**
  - Complex application
  - Complex data structure
Example 2: Adaptive Shadow Maps

- Fernando et al., ACM SIGGRAPH 2001
- Elegant solution to shadow map aliasing
  - Quadtree of small shadow maps
  - Shadow maps need resolution only on shadow boundary
  - Required resolution determined by projected area of screen space pixel into light space
Adaptive Shadow Maps

Why Adaptive Shadow Maps with Glift?

- Many recent (2004) shadow papers cite ASMs as high quality solution but not possible on graphics hardware.
- Algorithm is simple. Data structure is hard.
Adaptive Shadow Map Algorithm

- **Iterative refinement algorithm**
  - Identify shadow pixels with resolution mismatch
  - Create small shadow map “pages” at requested resolution

- **Shadow lookup**
  - Compute shadow map coordinate and resolution
  - Lookup in ASM (tree of small shadow map pages)

- **ASM depends on both camera and light position!**
ASM Data Structure Requirements

- Adaptive
- Multiresolution
- Fast, parallel random-access read
  - 2x2 native Percentage Closer Filtering (PCF)
  - Trilinear interpolated mipmapped PCF
- Fast, parallel write
- Fast, parallel insert and erase
ASM Data Structure

- Start with page table address translator
  - Coarse, uniform discretization of virtual domain
    - \(O(N)\) memory \(O(1)\) insert
    - \(O(1)\) computation \(O(1)\) erase
    - Uniform consistency
    - Partial mapping (sparse)
ASM Data Structure

• Page table example

vpn = va / pageSize

ppa = pageTable(vpn)
off = va % pageSize
pa  = ppa + off
ASM Data Structure Requirements

- Adaptive
- Multiresolution
- Fast, parallel random-access read
  - 2x2 native Percentage Closer Filtering (PCF)
  - Trilinear interpolated mipmapped PCF
- Fast, parallel write
- Fast, parallel insert and erase
ASM Data Structure

- Adaptive Page Table
  - Map multiple virtual pages to single physical page

\[
\begin{align*}
\text{vpn} &= \text{va} / \text{pageSize} \\
\text{ppa} &= \text{pageTable(vpn)}.\text{ppa()} \\
\text{s} &= \text{pageTable(vpn)}.\text{s()} \\
\text{off} &= (\text{va} \times \text{s}) \mod \text{pageSize} \\
\text{pa} &= \text{ppa} + \text{off}
\end{align*}
\]
ASM Data Structure Requirements

- Adaptive
- Multiresolution
- Fast, parallel random-access read
  - 2x2 native Percentage Closer Filtering (PCF)
  - Trilinear interpolated mipmapped PCF
- Fast, parallel write
- Fast, parallel insert and erase
ASM Data Structure

- Multiresolution Page Table

Virtual Domain

Mipmap Page Table

Physical Memory
ASM Data Structure Requirements

- Adaptive
- Multiresolution
- Fast, parallel random-access read
  - 2x2 native Percentage Closer Filtering (PCF)
  - Trilinear interpolated mipmapped PCF
- Fast, parallel write
- Fast, parallel insert and erase
ASM Data Structure Requirements

- How support bilinear filtering?
  - Duplicate 1 column and 1 row of texels in each page

- Mipmapped trilinear?
  - “By-hand” interpolation between mipmap levels
ASM Data Structure Requirements

- **Adaptive**
- **Multiresolution**
- **Fast, parallel random-access read**
  - 2x2 native Percentage Closer Filtering (PCF)
  - Trilinear interpolated mipmapped PCF
- **Fast, parallel write**
- **Fast, parallel insert and erase**
How Define ASM Structure in Glift?

- Start with generic page table `AddrTrans`
  - Use mipmapped `PhysMem` for page table
  - Change template parameter to add adaptivity
- Write page allocator
  - `alloc_pages`, `free_pages`
- Finally...

```cpp
typedef PageTableAddrTrans<...> PageTable;
typedef PhysMemGPU<vec2f, vec1s> PMem2D;
typedef VirtMemGPU<PageTable, PMem2D> VPageTable;
typedef AdaptiveMem<VPageTable, PageAllocator> ASM;
```
ASM Data Structure Usage

```c
float4 main( uniform VMem2D asm,
             float3 shadowCoord,
             float4 litColor ) : COLOR
{
    float isInLight = asm.vTex2Ds( shadowCoord );
    return lerp( black, litColor, isInLight );
}

asm.bind_for_read( ... );
asm.bind_for_write( ... );
asm.alloc_pages( ... );
asm.free_page( ... );
...
```
Adaptive Shadow Map Algorithm

- Faithful to Fernando et al. 2001
- Refinement algorithm
  - Identify shadow pixels with resolution mismatch (GPU)
  - Compact pixels into small stream (GPU)
  - CPU reads back compacted stream (GPU → CPU)
  - Allocate pages
    - Draw new PTEs into mipmap page tables (CPU → GPU)
    - Draw depth into ASM for each new page (GPU)
ASM: Effective resolution $131,072^2$ (37 MB); SM: $2048^2$

[Thanks to Yong Kil for the tree model]
“Octree” 3D Paint

- Interactive painting on unparameterized 3D surfaces
- 3D version of ASM data structure

**Differs from previous work:**
- Quadrilinear filtering
- $O(1)$, uniform access

- Interactive with effective resolutions between $64^3$ and $2048^3$
Demo
ASM Results

• Effective shadow map resolution up to $131,072^2$
  - $16^2$ - $64^2$ page size
  - $512^2$ - $2048^2$ page table
  - $2048^2$ - $4096^2$ physical memory
  - 20 - 80 MB

• Performance (45k polygon model)
  - 15 fps while moving camera (including refinement)
  - 5-10 fps while moving light

• Lookup time compared to $2048^2$ shadow map:
  - Bilinear filtered: 90% performance of traditional
  - Trilinear filtered mipmapped: 73%
Glift Results

- **Static instruction results**
  - With Cg program specialization

<table>
<thead>
<tr>
<th></th>
<th>Glift</th>
<th>By-Hand</th>
<th>Brook</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D → 2D</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3D page table</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ASM</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Octree</td>
<td>10</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>ASM + offset</td>
<td>10</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

- **Conclusion** : Glift structures within 1 instr of hand-coded Cg

Measured with NVShaderPerf, NVIDIA driver 75.22, Cg 1.4a
Overview

- Motivation and previous work
- Abstraction
- Implementation
- Examples
- Conclusions
Summary

- GPU programming needs data structure abstraction
  - Separate data structures and algorithms
  - More complex data structures and algorithms

- Why programmable address translation?
  - Common pattern in GPU data structures
  - Small amount of code virtualizes GPU memory model
Summary

• **Glift template library**
  - Generic C++/Cg implementation of abstraction
  - Nearly as efficient as hand coding
  - Integrates with OpenGL/Cg

• **Iterator computation model**
  - Generalize GPU computation model
  - Can future rasterizer increment iterators?
# Acknowledgements

- Craig Kolb, Nick Triantos, Cass Everitt  
  NVIDIA
- Fabio Pellacini  
  Dartmouth
- Adam Moerschell, Yong Kil  
  UCDavis
  Serban Porumbescu, Chris Co, ....
- Ross Whitaker, Chuck Hansen, Milan Ikits  
  U. of Utah

- National Science Foundation Graduate Fellowship
- Department of Energy
More Information

- Upcoming paper in ACM Transactions on Graphics
  - “Glift: Generic, Efficient, Random-Access GPU Data Structures”

- ACM SIGGRAPH 2005 Sketches
  - “Dynamic Adaptive Shadow Maps on Graphics Hardware”
  - “Octree Texture on Graphics Hardware”

- Google “Glift”