Compute Shaders 4

Motivation

- Go faster!

Recall

- Recall our benchmarks for raytracing triangle mesh:
  - CPU (basic algorithm): 8.60 sec/frame
  - CPU (octrees): 0.090 sec/frame
  - GPU (basic algorithm): 0.043 sec/frame
- CPU + acceleration structure is almost as good as GPU!
- Can we use an acceleration structure on the GPU?

Acceleration

- We designed our octree traversal algorithm to be nonrecursive, so that's not an issue
- But: Recall the definition of an octree node:

```
class OctreeNode{
  public:
    vec3 min, max;      //bounding volume; only used when
       building
    std::array<unsigned,8> children;    //0=no child
    std::vector<Triangle> triangles;
    std::array<vec4,6> planes;
    static std::vector<OctreeNode> nodes;
    ...
};
```

Problem

- The problem we'll face is that an octree node has a variable-length data structure: The triangles list
  - All the rest are fixed-size and thus easy to convert

Recall

- What data do triangles have?

```
class Triangle{
  public:
    vec3 p[3];      //vertices
    vec3 N;         //normal, unit length
    float D;            //plane equation D
};
```

- GLSL:

```
struct Triangle {
    vec4 p[3];
    vec4 ND;        //xyz=normal, w=D
};
layout(std430,binding=0) buffer Oct {
    octreeNodes[];
};
```

Octree Node

- **Octree node: In GLSL:**

```
struct OctreeNode {
    uint children[8];         //indices in octreeNodes
    ??? triangles;            //what to do here?
    vec4 planes[6];     //ABCD of each plane
};
//notice: binding = 1!!!
layout(std430,binding=1) buffer Oct {
    octreeNodes[];
};
```

Pattern

- We can use a pair of integers to hold “pointers” to a list of triangles
  - First pointer = index in the list of first triangle
  - Second = index past last triangle
- So we have:

```
struct OctreeNode {
    uint children[8];          //indices in octreeNodes
    uint firstTriangle;      //index in triangles[]
    uint lastTriangle;
    vec4 planes[6];          //ABCD of each plane
};
```

Note

- Note that CPU side code needs to rearrange triangles so they are contiguous
  - If we just use the Mesh data as it comes in: Won't work!
- Example updated code: Octree.h

Compute Shader

- Now we can incorporate the octree code in the compute shader:

```
bool traceOctree(vec3 s, vec3 v, out vec3 ip, out vec3 N, out vec3 color ){
    float closestT = -1.0;
    bool inter=false;
    uint stk[128];
    uint top=0;
    stk[top++] = 0; //push 0
    while( top != 0 ){
        uint ni = stk[--top];   //pop
        OctreeNode node = octreeNodes[ni];
        if( rayBoxIntersection( node.planes, s, v ) ){
            if( node.firstTriangle == node.lastTriangle ){
                for(int i=0;i<8;++i){
                    uint ci = node.children[i];
                    if( ci != 0 )
                        stk[top++] = ci;    //push(ci)
                }
            } else {
                //leaf
                if( traceTriangles( node.firstTriangle, node.lastTriangle, s,v,ip,N,closestT,color) )
                    inter=true;
            }
        }
    }
    return inter;
}
```

traceTriangles

- Almost identical to what we had before:

```
bool traceTriangles(uint first, uint last, vec3 s, vec3 v, out
    vec3 ip, out vec3 N,
      inout float closestT, out vec3 color )
{
    bool inter=false;
    for(uint i=first;i<last;++i){
        Triangle T = triangles[i];
        ...do intersection tests...
    }
}
```

Results

- CPU (basic algorithm): 8.60 sec/frame
- CPU (octrees): 0.090 sec/frame
- GPU (basic algorithm): 0.043 sec/frame
- GPU (Intel HD 5000): 0.011 sec/frame
- GPU (GeForce 920M): 0.024 sec/frame

Memory Organization

- Up to now, we haven't paid much attention to GPU storage types
- But it actually can make a big difference!
- What kind of storage is there?

Textures

- Textures
  - Potentially large amount of storage: Usually at least 4096x4096 (=16M texels)
  - Read-only (if attached to sampler) or Read/write (if attached to image unit)
  - Number available: Fairly high for samplers (Modern hardware = 96); fairly low for image units (8)
  - Homogeneous type; scalar or vec{2,3,4}
  - Comparatively slow to access (although GPU caching helps sequential access)

Buffers

- Buffers
  - Larger amount of storage: Often, up to entire size of GPU RAM (ex: Gigabyte range)
  - Read/write
  - Limited number available (8)
  - Can use structures to define heterogeneous types
  - Comparatively slow to access (although GPU caching helps sequential access)

Uniforms

- Uniforms
  - Limited space: Maybe as small as 16KB
  - Read only
  - Can use structures to define heterogeneous types
  - Fast access

Variables

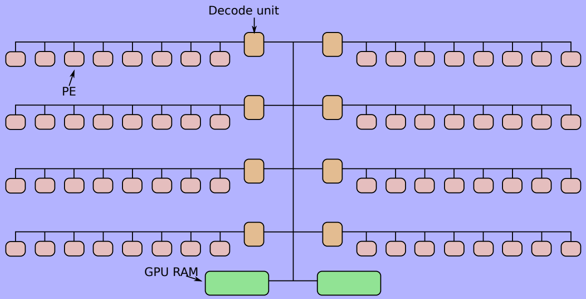
- Global/local variables
  - Number available: Depends on specific hardware
  - Read/write
  - Private to one shader invocation
  - Fast access

Problem

- We’ve been using buffers for our data
  - We probably have too much for uniforms
- But: Buffers aren’t the fastest way to access memory

Architecture

- Recall GPU architecture:



Texture/Buffer

- Textures and buffers sit in global RAM
- All the PE's contend for access to it
- And: It's not designed to be exceptionally fast (latency)
- This can be a problem for us...

Code

- Ignoring octrees, let's go back to the original GPU raytracing code
- What does it look like?

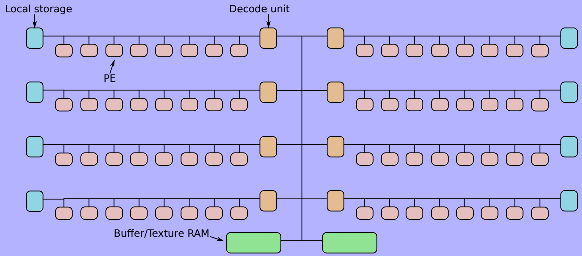
```
bool traceTriangles(vec3 s, vec3 v, out vec3 ip, out vec3 N,
            out vec3 color ){
    bool inter=false;
    float closestT = -1.0;
    int numTris = triangles.length();
    for(int i=0;i<numTris;++i){
        Triangle T = triangles[i];
        float denom = dot(T.ND.xyz,v);
        ...etc...
    }
}
```

Observe

- The access to triangles[] is not very fast (compared to computations)
  - Remember, it's stored in a buffer object
- And *all* the thread invocations are contending for access to this buffer
- It seems like we could improve this somehow...
  - ...We can!

Architecture

- A closer look at GPU architecture:



Memory

- Each workgroup has a local private memory area
- We can take advantage of this by sharing buffer data between workers

Approach

- First, we declare a global variable that's shared between the workgroups:
  shared Triangle cachedTriangles[64];
- Next, we adjust traceTriangles:
  - Each worker then copies exactly one thing from the triangles buffer to the cache
  - Then all the workers do ray intersections with items in the cache
  - Since we have more triangles than workers, we repeat the above steps until all triangles have been processed

ID

- gl_LocalInvocationID will give us our workgroup ID number
- Workgroup size was defined at the very top of the CS. Suppose we had something like this:

```
#define WORKGROUP_SIZE 64
layout(local_size_x=WORKGROUP_SIZE,local_size_y=1,local_size_z=1)
   in;
```

- Note: Before, we had size hardcoded at something like 64

Code

- We get something like this:

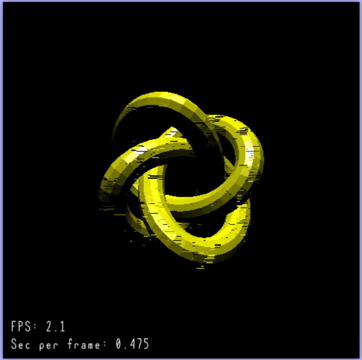
```
bool traceTriangles(vec3 s, vec3 v, out vec3 ip, out vec3 N, out vec3 color ){
    uint myId = gl_LocalInvocationID.x;
    bool inter=false;
    float closestT = -1.0;
    int numTris = triangles.length();
    for(int k=0;k<numTris;k+=WORKGROUP_SIZE){
        int last = k+WORKGROUP_SIZE;
        if( last > numTris )
            last = numTris;
        int numToCopy = last-k;
        if( myId < numToCopy ){
            cachedTriangles[myId] = triangles[k+myId];
        }
        for(int i=0;i<numToCopy;++i){
            Triangle T = cachedTriangles[i];
            float denom = dot(T.ND.xyz,v);
            ...use T...
        }
    }
}
```

Results

- Intel HD 5500:
  - No shared memory cache: 0.576 sec/frame
  - Shared memory cache: 0.475 sec/frame (1.2x speedup)
- GeForce 920M:
  - No shared memory cache: 0.291 sec/frame
  - Shared memory cache: 0.365 sec/frame (Surprise! It slowed down!)
- But there's a problem...

Problem

- The display is corrupted. Why?



Problem

- We don't know that all threads of workgroup proceed in lockstep
  - It's "mostly" simultaneous
  - But some workers can get held up (ex: Data fetch from buffer memory is delayed)
- Other workers continue on, assuming all data has been put in shared buffer

Solution

- We need to insert *barrier* to tell all threads to wait until everyone has arrived at a certain point

```
for(int k=0;k<numTris;k+=WORKGROUP_SIZE){
    int last = k+WORKGROUP_SIZE;
    if( last > numTris )
        last = numTris;
    int numToCopy = last-k;

    barrier();
    if( myId < numToCopy ){
        cachedTriangles[myId] = triangles[k+myId];
    }
    groupMemoryBarrier();
    barrier();
```

Explanation

- barrier() halts execution until all threads have reached that point
- groupMemoryBarrier ensures any writes prior to that point are visible to all other threads
- It makes sense why we'd need these two after the updating of cachedTriangles
- But why do we need barrier() *before* we write to cachedTriangles?

Explanation

- Remember, this is in a loop (the for-k loop)
- Suppose thread A completes the loop and cycles around for the next k-loop iteration
- If other threads are still working, they will assume that cachedTriangles isn't going to change out from under them
- But if A could just proceed on, that's exactly what would take place!
  - Result: We still get screen corruption
- Thus, we need barrier before and after.
  - If there was no for-k loop then we'd only need the post-barrier, not the pre-barrier
  - In my tests, adding the barrier gave correct display and only slowed rendering down by a few msec per frame

Sources

- https://www.khronos.org/opengl/wiki/ Memory_Model#Incoherent_memory_access
- https://www.khronos.org/opengl/wiki/ Synchronization#Implicit_synchronization
- https://www.khronos.org/opengl/wiki/ Buffer_Object_Streaming
- https://www.khronos.org/opengl/wiki/ Buffer_Object
- http://webglstats.com/
- https://stackoverflow.com/questions/7954927/passing-a-list-of-values-to-fragment-shader

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org