Compute Shaders 2

Review

- Purpose
- Coding
  - C++ / GLSL

Motivation

- We’ve seen one way to get data to/from GPU: Textures (Images)
  - Well suited for image-based operations
  - Easy to use and integrate with existing render pipeline
- But some important limitations
  - Only a limited number of image units available
    - Often, it’s limited to 8
  - Only suited for homogeneous data
    - Limited selection: float, vec{2,3,4}
  - Not easy to read back data to CPU

Buffers

- We can solve some issues with buffers
- Usually, buffers used for specifying vertex/index data
- But they are more flexible than that!

Buffers

- Recall: GPU has several buffer binding points
  - Attach buffer to binding point, work with it
  - Some binding points are indexed: Can attach different buffers to different indices
- Ex: For drawing
  - Attach buffer with vertex data to GL_ARRAY_BUFFER binding point
  - Attach buffer with vertex indices to GL_ELEMENT_ARRAY_BUFFER binding point
- For compute shaders, most interesting binding point is GL_SHADER_STORAGE_BUFFER

Usage

- We have framework for buffers in Buffer.h file
- If we want to communicate data CPU $\leftrightarrow$ GPU: Create *mappable* buffer:
- But there are (as usual) some complications...

Code

- Compute shader that does (very simple) buffer operation:

```
layout(local_size_x=64,local_size_y=1,local_size_z=1) in;
layout(std430,binding=0,row_major) buffer Foo {
    float buff[];
};
void main(){
    ivec3 mynum = ivec3(gl_GlobalInvocationID);
    buff[mynum[0]] = mynum[0]*10;
}
```

- Note buffer declaration syntax
  - std430 = Packing rules
  - binding = Which binding point
  - row_major = For matrices

Code

- Now for the CPU code: main.cpp
- When I run it, I get this output:
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  - That doesn't look right!

Problem

- GPU hasn't finished when CPU tries to access memory
- We need to synchronize the two sides
- Better code: main.cpp
- Now I get this output:
  0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150
- Much better!

Explanation

- glMemoryBarrier(GL_ALL_BARRIER_BITS)
  - This tells GL to insert a *memory barrier:* Things written before the barrier will be visible on the CPU when we *sync* on the data
- auto sync = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE,0);
  - This creates a *fence* object
  - Fences allow us to partition commands into two groups: One for each side of the fence
- auto rv = glClientWaitSync(sync, GL_SYNC_FLUSH_COMMANDS_BIT, -1);
  - This tells CPU to wait if the fence has not been reached by GPU
  - Third argument = timeout (nanoseconds, 64 bit unsigned value)
- glDeleteSync(sync);
  - Frees resources used by the fence

Note

- We don't need glMemoryBarrier and fence sync unless we plan to read back data *from* GPU *to* CPU
- If all data stays on GPU, no need for these

Structured Buffers

- Inconvenient to have only primitive types in buffers
- Example: Suppose we are processing collection of spheres
- Each sphere has:
  - Center (vec3)
  - Radius (float)
  - Color (vec4)
  - Reflectivity (float)

SoA

- We could code using "Structure of Arrays" (SoA) format:

```
layout(binding=0) buffer Foo {
    vec3 center[];
};
layout(binding=1) buffer Foo2 {
    float radius[];
};
layout(binding=2) buffer Foo3 {
    vec4 color[];
};
layout(binding=3) buffer Foo4 {
    float reflectivity[];
};
```

SoA

- To process a sphere:

```
uint idx = ...;
vec3 c = center[idx];
float r = radius[idx];
vec4 co = color[idx];
float ref = reflectivity[idx];
...use variables...
```

Problem

- One problem: Data for one sphere is “spread out” over memory
  - When we load data (CPU side): Must distribute it to different buffers
  - This may require extra time for conversion / data copying
- Also: We’re limited in how many buffers we can have active at once
  - Many GPU’s only permit 8
- We can use AoS format to get around this

Structs

- Defining structs: We need to do this in GLSL and in C
- Ex:

```
//GLSL
struct Sphere {
    vec4 center;
    float radius;
    vec4 color;
    uint reflectivity;
};
layout(std430,binding=0) buffer Foo {
    Sphere spheres[];
}
void main(){
    ...
    vec3 c = spheres[idx].center.xyz;
}
```

Padding

- GPU keeps data padded
- Alignment: A variable of a given type has an alignment (often, but not always, equal to its size)
  - Floats, ints, uints = 4
  - Doubles = 8
  - vec2, ivec2, uvec2 = 8
  - dvec2 = 16
  - vec3, ivec3, uvec3, vec4, ivec4, uvec4 = 16
  - dvec3, dvec4 = 32
- Arrays of items have each item one after another
- An MxN matrix is treated as an M element array of vecN's

Padding

- If an object would land at an impermissible address, it gets pushed further along until its address is OK
- In the case of the entire struct, it will have trailing padding so the size of the struct is a multiple of the largest n of any element in the struct
- Some examples may help...

Example

- GLSL declaration:

```
struct Foo {
    int a;
    int b;
};
```

- Memory layout:
  - 4 bytes for a
    - 4 bytes for b
  - Total size of structure: 8 bytes
    - No trailing padding

- C++ declaration:

```
struct Foo {
    alignas(4) int32_t a;
    alignas(4) int32_t b;
};
```

Example

- GLSL declaration:

```
struct Foo {
    float a;
    double b;
};
```

- C++ declaration:

```
struct Foo {
    alignas(4) float a;
    alignas(8) double b;
};
```

- Memory layout:
  - 4 bytes for a
    - 4 bytes of padding
    - 8 bytes for b
  - Total size of structure: 16 bytes
    - No trailing padding

Example

- GLSL declaration:

```
struct Foo {
    double a;
    uint b;
};
```

- Memory layout:
  - 8 bytes for a
    - 4 bytes for b
    - Total size: 12 bytes
    - Largest n: 8
    - So 4 bytes of padding
  - Total size of structure: 16 bytes

- C++ declaration:

```
struct Foo {
    alignas(8) double a;
    alignas(4) uint32_t b;
};
```

Example

- GLSL declaration:

```
struct Foo {
    int a[3];
    int b[4];
};
```

- C++ declaration:

```
struct Foo {
    alignas(4) int32_t a[3];
    alignas(4) int32_t b[4];
};
```

- Memory layout:
  - 12 bytes for a (4 per int)
    - 16 bytes for b (4 per int)
  - Total size of structure: 28 bytes
    - No padding

Example

- GLSL declaration:

```
struct Foo {
    double a[3];
    int b[4];
};
```

- C++ declaration:

```
struct Foo {
    alignas(8) double a[3];
    alignas(4) int32_t b[4];
};
```

- Memory layout:
  - 24 bytes for a (8 per double)
    - 16 bytes for b (4 per int)
  - Total size of structure: 40 bytes
  - No padding needed

Example

- GLSL declaration:

```
struct Foo {
    double a[3];
    int b;
};
```

- C++ declaration:

```
struct Foo {
    alignas(8) double a[3];
    alignas(4) int32_t b;
};
```

- Memory layout:
  - 24 bytes for a (8 per double)
    - 4 bytes for b
    - Total size = 28. Largest n = 8
    - So 4 bytes for padding
  - Total size of structure: 32 bytes

Example

- GLSL declaration:

```
struct Foo {
    vec4 a;
    int b;
};
```

- C++ declaration:

```
struct Foo {
    alignas(16) vec4 a;
    alignas(4) int32_t b;
};
```

- Memory layout:
  - 16 bytes for a (4 per element)
    - 4 bytes for b
    - Total = 20. Largest alignment n = 16 (for the vec4).
    - 12 bytes of padding to make it 32
  - Total size of structure: 32 bytes

Example

- GLSL declaration:

```
struct Foo {
    dvec4 a;
    int b;
};
```

- C++ declaration:

```
struct Foo {
    //math3d doesn't have dvec
    alignas(8) double a[4];
    alignas(4) int32_t b;
};
```

- Memory layout:
  - 32 bytes for a (8 per element)
    - 4 bytes for b
    - Total size: 36
    - Largest n = 4*8 = 32 (from the dvec4)
    - Next highest multiple of 32 is 64
    - Thus, 28 bytes of padding (36+28=64)
  - Total size of structure: 64 bytes

Example

- GLSL declaration:

```
struct Foo {
    int a;
    dvec4 b;
};
```

- Memory layout:
  - 4 bytes for a
  - 28 bytes of padding
  - 32 bytes for b
  - Total size: 64. No trailing padding.
- Total size of structure: 64 bytes

- C++ declaration:

```
struct Foo {
    alignas(4) int32_t a;
    alignas(8) double b[4];
};
```

Example

- GLSL declaration:

```
struct Foo {
    int a;
    dvec4 b;
    int c;
};
```

- C++ declaration:

```
struct Foo {
    alignas(4) int32_t a;
    alignas(32) double b[4];
    alignas(4) int32_t b;
};
```

- Memory layout:
  - 4 bytes for a, 28 bytes of padding, then 32 bytes for b, 4 bytes for c
  - Total size: 4+28+32+4 = 68
  - Largest n = 32 (for the dvec4). Next highest multiple of 32 is 96
  - So (96-68) = 28 bytes of trailing padding
- Total size of structure: 96 bytes

Example

- GLSL declaration:

```
struct Foo {
    vec3 a;
    mat3x2 b;
};
```

- C++ declaration:

```
struct Foo {
    alignas(16) vec3 a;
    alignas(8) vec2 b[3];
};
```

- Memory layout:
  - 12 bytes for vec3
  - 4 bytes of padding
  - b is treated as if it were declared as vec2 b[3]
  - vec2's must start on an 8 byte boundary
  - Then 24 bytes for b (8 bytes per vec3, 3 of them)
  - Total: 40 bytes. Largest n = 16 (for the vec3), so must have 8 more bytes of padding to make 48.

Example

- GLSL declaration:

```
struct Foo {
    vec3 a;
    mat3 b;
};
```

- C++ declaration:

```
struct Foo {
    alignas(16) vec3 a;
    alignas(16) vec4 b[3];  //
        Notice vec4!
};
```

- Memory layout:
  - 12 bytes for a, 4 bytes of padding, 48 bytes for b
    - Each vec3 takes up 3*4 = 12 bytes
    - 4 bytes of padding after each so next vec3 starts on address divisible by 16
  - Total: 64 bytes. No trailing padding needed.

Example

- Going back to the sphere...

```
//GLSL
struct Sphere {
    vec4 center;
    float radius;
    vec4 color;
    float reflectivity;
};
```

CPU

- Going back to the sphere:

```
struct Sphere {
    alignas(16) vec4 center;
    alignas(4) float radius;
    alignas(16) vec4 color;
    alignas(4) float reflectivity;
};
```

One More Example

- Suppose we have some sort of particles with position and velocity
- GLSL:

```
struct ParticleData {
    vec3 position;
    vec3 velocity;
};
```

- C:

```
struct ParticleData {
    alignas(16) vec3 position;
    alignas(16) vec3 velocity;
};
```

Example

- Now we can see an example of using CS to do some more intensive work
- Example: Sphere-based raytracer

Code

- Start with the GL framework code
- Replace some files:
  - draw.h
  - Globals.h
  - setup.h
  - shaders/cs.txt
- If you run this, you should get a solid magenta screen

Code

- We have several things we need to do:
  - Get the spheres to the compute shader
  - Do the raytracing in the compute shader

Data

- First we need to get the data to a buffer so the CS can use it
- Define a type at the top of setup.h:

```
struct GPUSphere{
    alignas(16) vec4 centerAndRadius;
    alignas(16) vec4 color;
};
```

Data

- Add some code to the bottom of setup():

```
std::vector<GPUSphere> sphereData(globs->scene.spheres.size());
for(unsigned i=0;i<globs->scene.spheres.size();++i ){
    sphereData[i].centerAndRadius = vec4(
        globs->scene.spheres[i].c, globs->scene.spheres[i].r);
    sphereData[i].color = vec4( globs->scene.spheres[i].color, 1.0
        );
}
auto b = Buffer::create(sphereData);
b->bindBase(GL_SHADER_STORAGE_BUFFER, 0);
```

Data

- Also set some uniforms in setup():

```
Program::setUniform("lightPosition", globs->scene.lightPosition);
Program::setUniform("lightColor", vec3(1,1,1) );
```

CS

- In the CS, we define the Sphere type and the buffer that holds the data:

```
struct Sphere {
    vec4 centerAndRadius;
    vec4 color;
};
layout(std430,binding=0) buffer S {
    Sphere spheres[];
};
```

Draw

- We change the draw.h file: Dispatch work to CS, then blit result to screen
- Code: [draw.h]

CS

- In CS, we do exactly one ray per invocation. Outline:

```
void main(){
    ivec2 pixelCoord = ivec2(gl_GlobalInvocationID.xy);
    ivec2 picSize = imageSize(img).xy;
    float d = 1.0 / 0.69974612;     //denom = tan(35 degrees) = field of view
    float dy = 2.0/(picSize.y-1);
    float dx = 2.0/(picSize.x-1);
    float y = -1.0 + pixelCoord.y * dy;
    float x = -1.0 + pixelCoord.x * dx;
    vec3 rayDir = x*cameraRight + y*cameraUp + d*cameraLook;
    bool wasIntersection;
    vec3 ip, N, color;
    wasIntersection = traceSpheres(eyePos, rayDir, ip, N, color );
    if( !wasIntersection )
        color = vec3(0,0,0);
    else
        color = shadePixel( ip, N, color );
    imageStore(img, ivec3(pixelCoord,0), vec4(color,1.0) );
}
```

That’s It!

- That’s it!
  - (Oh, except for changing Globals to load scene3.txt instead)

Results

- For spheres (scene3.txt):
  - CPU raytracing (Core i7, 2.4GHz): 0.978 sec/frame
  - GPU raytracing (Intel HD 5500): 0.084 sec/frame
  - GPU raytracing (GeForce 920M): 0.043 sec/frame

Assignment

- Implement triangle-mesh raytracing on the GPU using a compute shader
- For reference, here's what I got for scene2.txt:
  - CPU: 8.60 sec/frame
  - GPU (Intel HD 5500): 0.575 sec/frame
  - GPU (GeForce 920M): 0.290 sec/frame

Sources

- https://www.khronos.org/opengl/wiki/ Shader_Storage_Buffer_Object
- https://www.khronos.org/opengl/wiki/ Memory_Model#Incoherent_memory_access
- https://www.khronos.org/opengl/wiki/ GLAPI/glDeleteSync
- https://www.khronos.org/opengl/wiki/ Sync_Object
- https://www.khronos.org/opengl/wiki/ Synchronization#Implicit_synchronization
- https://www.khronos.org/opengl/wiki/ GLAPI/glMapBufferRange
- https://www.khronos.org/opengl/wiki/ GLAPI/glBufferStorage
- https://www.khronos.org/opengl/wiki/ Buffer_Object_Streaming
- https://www.khronos.org/opengl/wiki/ Buffer_Object

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