Compute Shader 3

Motivation

- Working with data structures on the GPU

Motivating Application

- We’ve been using raytracing as our example application
- But we haven’t been taking advantage of some useful aspects of it

Example

- Shadow computation is difficult with polygon renderers
- It's an easy extension with raytracers:
  - Compute first ray-triangle (or ray-sphere) intersection point p
  - Fire another ray from p to light source
    - If it hits something: p is in shadow
    - Else, p is lit

Reflections

- Reflective objects: This is tricky with polygon rasterization
- With raytracers, we can do it like so:
  - Compute first ray-triangle (or ray-sphere) intersection point p
  - Compute reflection vector R (reflect (p-eye) around surface normal at p)
  - Fire ray from p with direction R
  - Mix color of the object and whatever the reflected ray hits

Transparency

- What about semi-transparent objects?
  - Compute first ray-triangle (or ray-sphere) intersection point p
  - Compute refraction vector R (use Snell's law)
  - Fire ray from p with direction R
  - Mix object color with color of whatever the new ray hits

Note

- We can have multiple interreflections (think of a house of mirrors)
- Or reflective and refractive objects (ex: glass)
- Every time we hit such an object, we spawn one or two new rays
- We can do it recursively or we can just maintain a list or stack (a “to-do” list) and add reflection, transmission, and shadow rays to it

Implementation

- First, we'll look at a CPU implementation of shadows

Code

- Top-level raytrace function:

```
void raytrace(Scene& scene, std::vector<std::vector<vec3>>& pic){
    unsigned w = pic[0].size(), h = pic.size();
    float d = 1.0 / tan(scene.camera.fov_radians);
    float dy = 2.0/(h-1), dx = 2.0/(w-1);
    float y=1.0;
    for(unsigned yi=0;yi<h;++yi,y-=dy){
        float x=-1.0;
        for(unsigned xi=0;xi<w;++xi,x+=dx){
            vec3 rayDir = normalize(x*scene.camera.right + y*scene
                .camera.up + d*scene.camera.look);
            pic[yi][xi] = computeRayColor(scene, scene.camera.eye,
                rayDir );
        }
    }
}
```

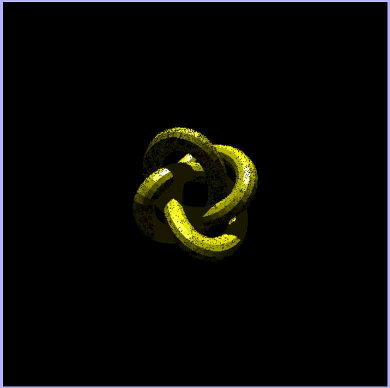
Code

- And then the computeRayColor function:

```
vec3 computeRayColor(Scene& scene, vec3& rayStart, vec3& rayDir ){
    vec3 ip,N,color;
    bool wasIntersection = traceRay(rayStart, rayDir, ip, N, color);
    if( !wasIntersection )
        return vec3(0,0,0);
    else{
        auto L = normalize(scene.lightPosition - ip);
        vec3 junk1,junk2,junk3; //don't need ip, normal, or color here
        wasIntersection = traceRay(scene, ip, L, junk1,junk2,junk3 );
        if( !wasIntersection ){
            return shadePixel( scene, rayStart, ip,N,color );
        } else {
            return 0.1 * color ;
        }
    }
}
```

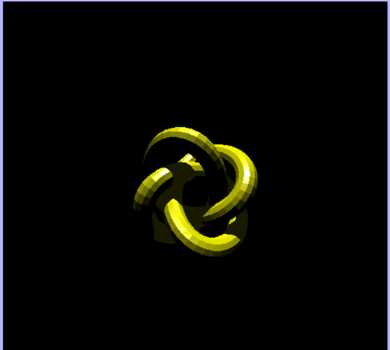
Problem

- When I run it, it looks...odd
- Why?



Solution

- The shadow ray hits the original surface immediately after starting!
  - We need to offset the ray by a small amount
  - In computeRayColor: Second call to traceRay:
    wasIntersection = traceRay(scene, ip + 0.01*L, L, junk1,junk2,junk3);



GPU

- Implementing on GPU: Easy: Just add another traceRay call when computing color
- Comparative results:
  - CPU: 8.59 sec/frame
  - GPU: 0.344 sec/frame

Reflective

- What about reflective objects?
- We just need to modify the computeRayColor function:

```
vec3 computeRayColor(Scene& scene, const vec3& rayStart, const vec3& rayDir, int bouncesLeft=5){
    vec3 ip,N,color;
    float alpha, refl, closestT;
    bool wasIntersection = traceRay(scene, rayStart, rayDir, ip, N, color, alpha, refl, closestT);
    if( !wasIntersection ){
        return vec3(0,0,0);
    } else{
        auto L = normalize(scene.lightPosition - ip);
        vec3 ip_, N_, color_;
        float alpha_, refl_, closestT_;
        auto inShadow = traceRay(scene, ip+0.01*L, L, ip_,N_,color_,alpha_,refl_,closestT_);
        vec3 baseColor;
        if( !inShadow )      baseColor = shadePixel( scene, rayStart, ip, N, color );
        else                 baseColor = 0.1 * color ;
        if( refl == 0.0 || bouncesLeft <= 0 )
            return baseColor;
        else{
            auto R = normalize( reflect( rayStart - ip, N ) );
            auto reflColor =  computeRayColor(scene, ip+0.01*R, R, bouncesLeft-1 );
            return baseColor + refl * reflColor;
        }
    }
```

Problem

- This function is recursive
- We can't implement recursion on the GPU (GLSL forbids it)
- What to do?
  - We could try making the function iterative

Code

```
vec3 computeRayColor(Scene& scene, vec3 rayStart, vec3 rayDir){
    vec3 ip,N,color,colorSoFar = vec3(0,0,0);;
    float alpha, refl, closestT, pctLeft=1.0;
    int bouncesLeft = 5;
    while(bouncesLeft > 0 ){
        bool wasIntersection = traceRay(scene, rayStart, rayDir, ip, N, color, alpha, refl, closestT);
        if( !wasIntersection ){
            return colorSoFar;
        } else{
            auto L = normalize(scene.lightPosition - ip);
            vec3 ip_, N_, color_, baseColor;
            float alpha_, refl_, closestT_;
            auto inShadow = traceRay(scene, ip+0.01*L, L, ip_,N_,color_,alpha_,refl_,closestT_);
            if( !inShadow ) baseColor = shadePixel( scene, rayStart, ip, N, color );
            else            baseColor = 0.1 * color ;
            colorSoFar = colorSoFar + pctLeft * baseColor;
            if( refl == 0.0 )
                bouncesLeft = 0;
            else{
                auto R = normalize( reflect( rayStart - ip, N ) );
                bouncesLeft--;
                rayStart = ip+0.01*R;
                rayDir = R;
                pctLeft = pctLeft * refl;
            }
        }
    }
    return colorSoFar;
}
```

GPU

- We could translate this more-or-less exactly to the GPU
- But: That might not be ideal
- Why not?

Problem

- Several problems with this approach:
  - If only some pixels are reflective, a few processors may get bogged down computing reflections when other processors are idle
  - What if we have reflection and refraction?
    - Each ray may spawn *two* rays
    - It's not clear how we could integrate that into our setup
  - If CS takes too long to complete, driver will assume it's locked up
    - Resets GPU, dumping all the work we've done

Solution

- Each CS invocation handles only one round of rays
- Reflected rays get added to a buffer for future processing

```
struct Ray{
    ???
};
layout(std430,binding=2,row_major) buffer InData2{
    int currentRayCount;
    int padding[3];
    Ray currentRays[];
};
layout(std430,binding=3,row_major) buffer InData3{
    int nextRayCount;
    int padding[3];
    Ray nextPassRays[];
};
```

Explanation

- currentRays:
  - This holds the set of rays we are going to cast on this round
  - currentRayCount holds the number of rays
  - The padding ensures currentRays starts on a 16 byte boundary
- nextPassRays:
  - Will receive set of rays to cast on next round
  - Also has count and padding.

Ray

- What data do we need for a ray?

Ray

- Ray will need to know:
  - Screen (pixel) coordinates associated with it
  - Starting point
  - Direction
- So:

```
struct Ray{
    vec4 start,dir,coords;
};
```

  - We only use two components of coords, but padding would add two extra floats if we declared as vec2

Render

- CPU must create two buffers for the rays
- Bind to each location (2 and 3)
- Dispatch CS to do the render

CS

- CS will pull a ray from currentRays
- Cast the ray, computing its color
- If it does not hit a reflective object:
  - Output color to the image
- Else
  - Compute reflection ray and put in nextPassRays

Idea

- We repeatedly dispatch CS
- Each time, things are taken out of nextPassRays and new things are put into it
- After some number of dispatches, we're done

Initialization

- Question: How to initialize currentRays when we start up?

CPU

- Possibility 1: Initialize on CPU

```
vector<RayInfo> V;
for(y=0;y<h;++y){
    for(x=0;x<w;++x){
        R = ...ray from eye through pixel (x,y)
        V.push_back(R);
    }
}
auto rayBuffer1 = Buffer::createMappable(V);
```

Analysis

- Fairly simple
- Drawback: Need to re-do every time camera moves
  - Consumes CPU to GPU bandwidth uploading data
- CPU could better spend its time doing something else (AI, network, user input...)
- GPU is idle waiting for CPU

Option 2

- Let the GPU do the initialization
- Define a uniform: bool firstPass
- CPU: Set firstPass to true
- Then dispatch CS
- CPU: Set firstPass to false
- Dispatch CS a second time
- Dispatch CS a third time
- Dispatch CS a fourth time
- ...etc...
- Ex: If we want max of 5 interreflections: Do five dispatches

CS

- CS looks like this:

```
void main(){
    if( firstPass ){
        uvec2 pixCoord = gl_GlobalInvocationID.xy;
        r = ray from eye through pixCoord
    } else {
        if( currentRays is empty )
            return;
        r = take item from currentRays
    }
    do ray cast using r
```

takeItem

- How to take item from currentRays?

```
bool takeItem(out Ray r){
    if( currentRayCount == 0 )
        return false;
    r = currentRays[--currentRayCount];
    return true;
}
```

- This is totally wrong. Why?

Concurrency

- We have to consider concurrency issues!
- Suppose workers A and B both take items at same time
  - With thousands of GPU cores working at once, this is a real possibility!
- A & B might both do the same ray
- Another Problem: Check-then-act
  - A might check, find there's data available, but B might take the last item before A gets a chance to retrieve that data item
- How to solve?

Atomics

- We can use atomic operations
- Example:

```
bool takeItem(out Ray r){
    int idx = atomicAdd(currentRayCount,-1);
    if( idx <= 0 ){
        atomicAdd(currentRayCount,1);
        return false;
    }
    r = currentRays[idx-1];
    return true;
}
```

Reflections

- Whenever we have ray intersect a reflective object, we add that reflected ray to the next round buffer

```
bool putItem(Ray r){
    int idx = atomicAdd(nextRayCount, 1);
    if( idx >= nextPassRays.length() ){
        atomicAdd(nextRayCount, -1);    //buffer is full; discard
            ray
        return false;
    }
    nextPassRays[idx] = r;
    return true;
}
```

CPU Code

- Repeatedly dispatch for as many levels of nested reflections as we need:

```
unsigned zero[] = {0,0,0,0};
std::shared_ptr<Buffer> bufferA = rayBuffer1;
std::shared_ptr<Buffer> bufferB = rayBuffer2;
for(int i=0;i<numPasses;++i){
    Program::setUniform("firstPass", i==0 );
    Program::updateUniforms();
    bufferA->bindBase(GL_SHADER_STORAGE_BUFFER,2);   //current rays
    bufferB->bindBase(GL_SHADER_STORAGE_BUFFER,3);   //next pass rays
    //clear next pass's ray count
    glClearBufferSubData(GL_SHADER_STORAGE_BUFFER,GL_R32UI,
        0,4,GL_UNSIGNED_INT, zero);
    prog.dispatch( w/64, h, 1 );
    swap(bufferA,bufferB);  //pointer swap
}
```

Problem

- It's all great...Except one problem
- We haven't put the colors anywhere!
- We can store those in the rays too

Idea

- CS has the image2D uniform for final image
- Rays have a field for “accumulated color” and one for “percentageRemaining”
- We use these like so...

Rays

- Suppose we take ray from buffer
  - Suppose its percentageRemaining == 1
- We cast ray
- It hits surface that is 25% reflective. Surface's color (after lighting) is c.

Ray

- Create new ray:
  - accumulatedColor = 0.75*c
  - percentageRemaining = 0.25
  - start = intersection point
  - direction = reflected direction
  - coordinates = input ray's coordinates
  - Don't change the image2D yet
- Add new ray using putItem()

Idea

- If we take ray from buffer where percentageRemaining is close to zero or else ray hits non-reflective surface or we're on last round of processing:
  - Write final ray color to image2D at coordinates specified by ray
  - Don't add anything to next round buffer
- Result: As multiple reflections occur, new surfaces contribute less and less to ray color
- Eventually, we stop reflecting
- Refracted rays could be handled in similar fashion

Assignment

- Implement ray shadows for polygonal objects using compute shaders
- For bonus [+100%], also implement reflective objects using the compute shader

Sources

- https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object
- https://www.khronos.org/opengl/wiki/Memory_Model#Incoherent_memory_access

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