Welcome!
Today's Agenda:

- Practical GPGPU: Verlet Fluid
- GPGPU Algorithms
- Optimizing GPU code
Verlet

https://www.youtube.com/watch?v=JcgkAMr9r5o
Verlet

Verlet Physics

Motion:

\[ x_1 = x_0 + v_0 \Delta t \]

We can express this without velocities:

\[ x_2 = x_1 + (x_1 - x_0) \]

Simulation:

- Backup current position: \( x_{\text{current}} = x \)
- Update positions: \( x = x + (x - x_{\text{previous}}) \)
- Store last position: \( x_{\text{prev}} = x_{\text{current}} \)
- Apply constraints (e.g. walls)

Applying constraints:

- e.g. if \( (x < 0) \) \( x = 0 \);
Verlet

Verlet Physics

Cloth:

- Using a grid of vertices
- Forces on all vertices: gravity
- Constraint for top row: fixed position
- Constraint for all vertices: maximum distance to neighbors

Fluid:

- Using large collection of particles
- Forces on all particles: gravity
- Constraint for all particles: container boundaries
- Constraint for all particles: do not intersect other particles
GPU Verlet Fluid

Input:
- Array of particle positions
- Array of previous particle positions

Output:
- Visualization of simulation
- Array of particle positions (updated)
- Array of previous particle positions (updated)
Verlet

GPU Verlet Fluid

STAGE 1

Drawing a number of moving particles using OpenCL
Buffer* balls = new Buffer( BALLCOUNT * 6 * sizeof( float ) );

// put initial ball positions in buffer
float* fb = (float*)balls->GetHostPtr();
for( int i = 0; i < BALLCOUNT; i++ )
{
    fb[i * 6] = Rand( 1 );
    fb[i * 6 + 1] = Rand( 1 );
    fb[i * 6 + 2] = Rand( 0.01f ) - 0.005f;
    fb[i * 6 + 3] = Rand( 0.01f ) - 0.005f;
    fb[i * 6 + 4] = fb[i * 6 + 0];
    fb[i * 6 + 5] = fb[i * 6 + 1];
}
balls->CopyToDevice();

position
velocity (for now)
**Verlet**

GPU Verlet Fluid – Device Code

```
__kernel void clear( write_only image2d_t outimg )
{
    int column = get_global_id(0);
    int line = get_global_id(1);
    if ((column >= 800) || (line >= 480)) return;
    write_imagef(outimg, (int2)(column, line), 0);
}

__kernel void update( global float* balls )
{
    int idx = get_global_id(0);
    balls[idx * 6 + 0] += balls[idx * 6 + 2];
    balls[idx * 6 + 1] += balls[idx * 6 + 3];
}
```

**Task:**
- write a single black pixel.

**Workset:**
- number of pixels.

**Task:**
- Update the position of one ball.

**Workset:**
- Number of balls.
__kernel void render( write_only image2d_t outimg, global float* balls )
{
    int column = get_global_id( 0 );
    int line = get_global_id( 1 );
    float2 uv = { (float)column / 800.0, (float)line / 480.0 };
    for( int i = 0; i < BALLCOUNT; i++ )
    {
        float2 pos = { balls[i * 6], balls[i * 6 + 1] };
        float dist = length( pos - uv );
        if (dist > 0.02f) continue;
        write_imagef( outimg, (int2)(column, 479 - line), (float4)(1,0,0,1) );
        break;
    }
}
GPU Verlet Fluid – Result
Verlet

GPU Verlet Fluid

Rendering many particles efficiently
GPU Verlet Fluid – Grid

Host:

```cpp
grid = new Buffer( GRIDX * GRIDY * (BALLSPERCELL + 1) * sizeof( unsigned int ) );
```

Device:

```cpp
__kernel void clearGrid( global unsigned int* grid )
{
    int idx = get_global_id( 0 );
    int baseIdx = idx * (BALLSPERCELL + 1);
    grid[baseIdx] = 0;
}
```

Data layout:

- [0]: ball count for cell
- [1..N]: ball indices

Task:

- Reset a grid cell by setting ball count to 0.

Workset:

- Number of cells.
__kernel void fillGrid( global float* balls, global unsigned int* grid )
{
    int ballIdx = get_global_id( 0 );
    int gx = balls[ballIdx * 6 + 0] * GRIDX;
    int gy = balls[ballIdx * 6 + 1] * GRIDY;
    if ((gx < 0) || (gy < 0) || (gx >= GRIDX) || (gy >= GRIDY)) return;
    int baseIdx = (gx + gy * GRIDX) * (BALLSPERCELL + 1);
    int count = grid[baseIdx]++;
    grid[baseIdx + count + 1] = ballIdx;
}

Task:
- Add a single ball to the correct grid cell.

Workset:
- Number of balls.
__kernel void fillGrid( global float* balls, global unsigned int* grid )
{
    int ballIdx = get_global_id( 0 );
    int gx = balls[ballIdx * 6 + 0] * GRIDX;
    int gy = balls[ballIdx * 6 + 1] * GRIDY;
    if ( ((gx < 0) || (gy < 0) || (gx >= GRIDX) || (gy >= GRIDY)) ) return;
    int baseIdx = (gx + gy * GRIDX) * (BALLSPERCELL + 1);
    unsigned int count = atomic_inc( grid + baseIdx );
    if ( count < BALLSPERCELL ) grid[baseIdx + count + 1] = idx; else
    {
        balls[ballIdx * 6 + 1] = balls[ballIdx * 6 + 5] = 0.1;
        grid[baseIdx] = BALLSPERCELL;
    }
}

Verlet

GPU Verlet Fluid – Grid
**Verlet**

**GPU Verlet Fluid – Grid**

```c
__kernel void render( write_only image2d_t outimg, global float* balls, 
global unsigned int* grid )
{
    int column = get_global_id( 0 );
    int line = get_global_id( 1 );
    if ((column >= 800) || (line >= 480)) return;
    float2 uv = { (float)column / 800.0, (float)line / 480.0 };
    // draw balls using grid
    int gx = uv.x * GRIDX;
    int gy = uv.y * GRIDY;
    int gx1 = max( 0, gx - 1 ), gx2 = min( GRIDX - 1, gx + 1 );
    int gy1 = max( 0, gy - 1 ), gy2 = min( GRIDY - 1, gy + 1 );
    ...
for ( int y = gy1; y <= gy2; y++ )
for ( int x = gx1; x <= gx2; x++ )
{
    unsigned int baseIdx = (x + y * GRIDX) * (BALLSPERCELL + 1);
    unsigned int count = grid[baseIdx];
    for ( int i = 0; i < count; i++ )
    {
        unsigned int ballIdx = grid[baseIdx + i + 1];
        float2 pos = { balls[ballIdx * 6], balls[ballIdx * 6 + 1] };
        float dist = length( pos - uv );
        if ( dist > 0.01f ) continue;
        write_imagef( outimg, (int2)(column, 479 - line), (float4)(1,0,0,1) );
    }
}
Verlet

GPU Verlet Fluid – Grid - Result
Verlet

GPU Verlet Fluid

Implementing simulation
Verlet

GPU Verlet Fluid – Simulation

```c
__kernel void simulate1( global float* balls )
{
    int idx = get_global_id( 0 );

    float2 prevPos = { balls[idx * 6 + 0], balls[idx * 6 + 1] };
    float2 delta = { balls[idx * 6 + 0] - balls[idx * 6 + 4],
                     balls[idx * 6 + 1] - balls[idx * 6 + 5] + 0.0002 };

    float speed = length( delta );
    if (speed > 0.01f) delta = 0.01f * normalize( delta );

    balls[idx * 6 + 0] += delta.x;
    balls[idx * 6 + 1] += delta.y;
    balls[idx * 6 + 4] = prevPos.x;
    balls[idx * 6 + 5] = prevPos.y;
}
```
__kernel void simulate2( global float* balls, global float* balls2, 
global unsigned int* grid )
{
    int cellIdx = get_global_id( 0 );
    int baseIdx = cellIdx * (BALLSPERCELL + 1);
    int count = grid[baseIdx];
    if (count == 0) return;
    int gx = idx % GRIDX;
    int gy = idx / GRIDX;
    // determine 3x3 block around current cell
    int gx1 = max( 0, gx - 1 ), gx2 = min( GRIDX - 1, gx + 1 );
    int gy1 = max( 0, gy - 1 ), gy2 = min( GRIDY - 1, gy + 1 );
    for( int i = 0; i < count; i++ )
    {
Verlet

GPU Verlet Fluid – Simulation

// get active ball
int idx1 = grid[baseIdx + i + 1];
float2 ball1Pos = { balls[idx1 * 6 + 0], balls[idx1 * 6 + 1] };

// evade other balls
for( int y = gy1; y <= gy2; y++ ) for( int x = gx1; x <= gx2; x++ )
{
    int baseIdx = (x + y * GRIDX) * (BALLSPERCELL + 1);
    int count2 = min( (unsigned int)BALLSPERCELL, grid[baseIdx] );
    for( int j = 0; j < count2; j++ )
    {
        int idx2 = grid[baseIdx + j + 1];
        if (idx2 != idx1)
            float2 ball2Pos = { balls2[idx2 * 6 + 0], balls2[idx2 * 6 + 1] };
    }
}
GPU Verlet Fluid – Simulation
Verlet

GPU Verlet Fluid

What causes the poor performance?

- Simulation handles one grid cell *per thread*
- Grid cell workload is highly irregular
- Do we even have enough grid cells?
Verlet

GPU Verlet Fluid - TakeAway

GPGPU is a bit different:

- We have 'host' and 'device' code
- We need many small identical tasks
- Each task has an 'identity' (1D, 2D or 3D index in the workset)
- Some tasks may be outside the workset (check for this!)
- Ideally, each of those tasks should do a similar amount of work (if, for)
- The tasks run in parallel: mind concurrency issues! (atomic)
- Data transfer from CPU to GPU is expensive (avoid this)

In this example, OpenCL directly plotted to an OpenGL texture (which is then drawn on a quad, using a shader). It is probably more efficient to let OpenCL prepare a vertex buffer for drawing point sprites.
Today's Agenda:

- Practical GPGPU: Verlet Fluid
- GPGPU Algorithms
- Optimizing GPU code
GPGPU Algorithms

Prefix Sum

The prefix sum (or cumulative sum) of a sequence of numbers is a second sequence of numbers consisting of the running totals of the input sequence:

Input: \( x_0, x_1, x_2 \)
Output: \( x_0, x_0 + x_1, x_0 + x_1 + x_2 \) (inclusive) or \( 0, x_0, x_0 + x_1 \) (exclusive).

Example:

<table>
<thead>
<tr>
<th>input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>inclusive</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>exclusive</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

Here, addition is used; more generally we can use an arbitrary binary associative operator.
GPGPU Algorithms

Prefix Sum

In C++:

```cpp
out[0] = 0;
for ( i = 1; i < n; i++ ) out[i] = in[i-1] + out[i-1];
```

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>inclusive</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>exclusive</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>
GPGPU Algorithms

Prefix Sum

The prefix sum is used for compaction.

Given: kernel $K$ which may or may not produce output for further processing.
GPGPU Algorithms

Prefix Sum - Compaction

Given: kernel K which may or may not produce output for further processing.
GPGPU Algorithms

Prefix Sum

\[
\text{out}[0] = 0; \\
\text{for } (i = 1; i < n; i++) \text{ out}[i] = \text{in}[i-1] + \text{out}[i-1]; \\
\]

In parallel:

\[
\text{for } (d = 1; d <= \log_2 n; d++) \\
\text{for all } k \text{ in parallel do } \text{ if } k \geq 2^{d-1} \text{ then } x[k] += x[k - 2^{d-1}] \\
\]
GPGPU Algorithms

Prefix Sum

\[
\text{out}[0] = 0; \\
\text{for} \ ( i = 1; i < n; i++ ) \ \text{out}[i] = \text{in}[i-1] + \text{out}[i-1];
\]

In parallel:

\[
\text{for} \ ( d = 1; d \leq \log_2 n; d++ ) \text{ for all } k \text{ in parallel do} \\
\text{if } k \geq 2^{d-1} \text{ then } x[k] += x[k - 2^{d-1}]
\]

Notes:

- The scan happens in-place. This is only correct if we have 32 input elements, and the scan is done in a single warp. Otherwise we need to double buffer for correct results.
- Time complexity of the algorithm is $O(n \log n)$. For large $n$, it is not work-efficient.
GPGPU Algorithms

Prefix Sum

Phase 1: “Up-sweep”

Phase 2: “Down-sweep”

Notes:
- A scan consists of $2(n-1)$ additions and $n-1$ swaps.

More information:
GPU Gems 3, Chapter 39, “Parallel Prefix Sum (Scan) with CUDA”, Mark Harris.
**GPGPU Algorithms**

**Prefix Sum**

You can find an implementation of the prefix sum in the OpenCL template:

```cpp
cl_int Buffer::ParallelScan()
```

This replaces the contents of a buffer with the prefix sum of the same buffer.
GPGPU Algorithms

GPU Sorting: Selection Sort

__kernel void Sort( __global int* in, __global int* out )
{
    int i = get_global_id( 0 );
    int n = get_global_size( 0 );
    int iKey = in[i];
    // compute position of in[i] in output
    int pos = 0;
    for( int j = 0; j < n; j++ )
    {
        int jKey = in[j]; // broadcasted
        bool smaller = (jKey < iKey) || (jKey == iKey && j < i);
        pos += (smaller) ? 1 : 0;
    }
    out[pos] = iKey;
}
GPGPU Algorithms

GPU Sorting
GPGPU Algorithms

GPU Sorting

Size: number of comparisons (in this case: $5 + 4 + 3 + 2 + 1 = 15$)
Depth: number of sequential steps (in this case: 9)
GPGPU Algorithms

GPU Sorting
GPGPU Algorithms

GPU Sorting

You can find an implementation of the bitonic sort in the OpenCL template:

```c
cl_int Buffer::ParallelSort()
```

This replaces the contents of a buffer with the sorted values.
GPGPU Algorithms

Take-away:

GPGPU requires massive parallelism. Algorithms that do not exhibit this need to be replaced.

The parallel scan is an important ingredient that serves as a building block for larger algorithms, or between kernels.
Today’s Agenda:

- Practical GPGPU: Verlet Fluid
- GPGPU Algorithms
- Optimizing GPU code
Optimizing GPGPU
Optimizing GPGPU
Optimizing GPGPU
1. Optimize memory usage
   - Read data from global memory once
   - Use local memory when possible
   - Careful: reading the same global address in 32 threads is not a good idea!

2. Make sure there is enough work to hide latency
   - On AMD: use multiples of 64 threads (called a ‘wavefront’)
   - Tweak manually for performance, ideally per vendor / device

3. Minimize the number of host-to-device transfers, then their size

4. Minimize the number of kernel invocations

Optimizing GPGPU

Faster OpenCL

Smaller things:

- Use float4 whenever possible
- Use predication rather than control flow
- Bypass short-circuiting
- Remove conditional code
- AOS vs SOA performance
- Reducing atomics
- Reduced precision math
- Pinned memory

```cpp
if (A>B) C += D; else C -= D;
Replace this with:
int factor = (A>B) ? 1:-1;
C += factor*D;
```

```cpp
if(a&&b&&c&&d){...}
becomes
bool cond = a&&b&&c&&d;
if(cond){...}
```

```cpp
if(x==1) r=0.5;
if(x==2) r=1.0;
becomes
r = select(r, 0.5, x==1);
r = select(r, 1.0, x==2);
```
Today's Agenda:

- Practical GPGPU: Verlet Fluid
- GPGPU Algorithms
- Optimizing GPU code
END of “GPGPU (2)”

next lecture: Presentations