Welcome!
Today's Agenda:

- Practical GPGPU: Verlet Fluid
- (in several steps)
Verlet
.INGREDIENTS
Verlet

Verlet Physics

Motion along a straight line:

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

For a fixed time step we can express this without explicit velocities:

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

Simulation:

- Backup current position: \( x_{current} = x \)
- Update positions: \( x = x + (x - x_{previous}) \)
- Store last position: \( x_{prev} = x_{current} \)
- 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
Verlet

Template

Texture:
- To efficiently display OpenCL output using OpenGL

Shader:
- As an alternative to OpenCL, e.g. for postprocessing

Kernel:
- Specifying actual device code:
- Setting and changing arguments:
- Launching the kernel:

```
fractal = new Kernel("programs/program.cl", "TestFunction");
fractal->SetArgument(0, outputBuffer);
fractal->Run(outputBuffer);
```
**Verlet**

**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

Idea:

Let’s draw 128 balls, brute force.

Data:

- Screen buffer, 800x480
- Ball data, 128 records

Procedure:

1. Clear screen
2. Update ball positions
3. Draw balls

Drawing balls, options:

- Loop over balls
- Loop over pixels

Check 128 balls per pixel
Verlet

GPU Verlet Fluid – Host Code

Buffer* balls = new Buffer( BALLCOUNT * 6 );
// 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

```c
__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;
    }
}
Verlet Fluid – Result
GPU Verlet Fluid

Rendering many particles efficiently

STAGE 2

Idea:

Let’s use a grid to reduce the number of balls we check per pixel.

Data:

- Grid, custom resolution
- Fixed room per cell for N balls

Procedure:

1. Clear grid
2. Add balls to grid
3. Render pixels
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;
    }
}

__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;
    }
}
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 );
    ...
```
Verlet

GPU Verlet Fluid – Grid

```c
... 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) );
    }
}
```
GPU Verlet Fluid – Grid - Result
Verlet

GPU Verlet Fluid

STAGE 3

Implementing simulation

Idea:

Basics work; let's add some physics.

Procedure:

1. Move particles
2. Satisfy constraints
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;
}
```
Verlet

GPU Verlet Fluid – Simulation

```c
__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++)
    {
        // ... (omitted for brevity)
    }
}
```
Verlet

GPU Verlet Fluid – Simulation

```c
// 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

INFOMOV – Lecture 10 – “GPGPU (2)”
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

STAGE 4

Improving performance

Idea:

*Grid cells are filled irregularly; loop over balls for simulation.*

Procedure, simulation:

1. A ball checks its surroundings in the grid.

Procedure, rendering (new):

- For rendering we loop over balls too. If two balls fight for the same pixel, we ignore that.
**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
- (in several steps)
END of “GPGPU (2)”

next lecture: GPGPU (3)