# Game Physics Demo

Berk Emre Saribas
Yunus Can Cakir

# Exercise 1 - Algorithm

1. Clear forces + add gravity + mouse force

```
for (auto& mp : mass_points) {
    mp.force = this->external_force + mouse_force;
}
```

# Exercise 1 - Algorithm

2. Compute elastic forces, add damping

```
void MassSpringSystemSimulator::compute_elastic_force(const Spring& s) {
    auto& mp1 = mass_points[s.mp1];
    auto& mp2 = mass_points[s.mp2];

    Vec3 spring_vec = mp1.position - mp2.position;
    float x = norm(spring_vec) - s.initial_length;
    normalize(spring_vec);
    Vec3 f = -stiffness * x * spring_vec;
    mp1.force += f;
    mp2.force += -f;
}
```

```
for (const auto& spring : springs) {
    compute_elastic_force(spring);
}

//damping
for (auto& mp : mass_points) {
    mp.force += -damping * mp.velocity;
}
```

# Exercise 1 - Algorithm

3. Integrate

```
switch (integrator) {
case EULER:
{
    for (auto& mp : mass_points) {
        Vec3 accel = mp.force / mass;
        if (!mp.is_fixed) {
            mp.position = mp.position + delta * mp.velocity;
        }
        mp.velocity = mp.velocity + delta * accel;
    }
    break;
}
case LEAPFROG:
{
    for (auto& mp : mass_points) {
        Vec3 accel = mp.force / mass;
        // Not tested
        mp.velocity = mp.velocity + delta * accel;
        if (!mp.is_fixed) {
            mp.position = mp.position + delta * mp.velocity;
        }
    }
    break;
}
```

# Exercise 1 - Algorithm

## 3. Integrate (Midpoint)

```
case MIDPOINT:
{
    old_positions.clear();
    old_velocities.clear();

    for (auto& mp : mass_points) {
        Vec3 accel = mp.force / mass;

        old_positions.push_back(mp.position);
        old_velocities.push_back(mp.velocity);

        if (!mp.is_fixed) {
            mp.position = mp.position + delta / 2.0f * mp.velocity;
        }
        mp.velocity = mp.velocity + delta / 2.0f * accel;
    }
```

```
for (auto& mp : mass_points) {
    mp.force = this->external_force + mouse_force;
}

for (const auto& spring : springs) {
    compute_elastic_force(spring);
}

for (auto& mp : mass_points) {
    mp.force += -damping * mp.velocity;
}

for (int i = 0; i < mass_points.size(); i++) {
    Vec3 accel = mass_points[i].force / mass;

    if (!mass_points[i].is_fixed) {
        mass_points[i].position = old_positions[i] + delta * mass_points[i].velocity;
    }
    mass_points[i].velocity = old_velocities[i] + delta * accel;
}
```

# Exercise 1 - 3D Cloth Scene

- Cloth as a grid
- Two implementations
  - CPU
    - 4 directions
  - GPU
    - Compute shader
    - 8 directions

# Exercise 1 - Collision (3D Cloth Scene)

```
void MassSpringSystemSimulator::compute_collision() {
    const float dn = 0.001;
    for(auto& mp : mass_points) {
        // Sphere collision
        auto dist_sphere = mp.position - sphere_pos;
        mp.colliding = false;
        if(norm(dist_sphere) < sphere_rad + dn) {
            mp.velocity = 0;
            normalize(dist_sphere);
            mp.position = sphere_pos + dist_sphere * (sphere_rad + dn);
            mp.colliding = true;
        }
```

```
        // Cube collision
        auto dist_cube = mp.position - cube_pos;
        auto cube_boundary = cube_rad + 20.0 * dn;
        auto clamped = clamp(dist_cube, -cube_boundary, cube_boundary);
        Vec3 axis;
        if(abs(clamped.x) > cube_rad) {
            axis[0] += sign(clamped.x);
        }
        if(abs(clamped.y) > cube_rad) {
            axis[1] += sign(clamped.y);
        }
        if(abs(clamped.z) > cube_rad) {
            axis[2] += sign(clamped.z);
        }
        if(norm(dist_cube - clamped) < dn) {
            mp.velocity = 0;
            mp.position += axis * dn;
            mp.colliding = true;
        }
        // Floor collision
        if(mp.position.y < -1.0f) {
            mp.position.y = -1.0f + dn;
            mp.velocity = 0;
            mp.colliding = true;
        }
}
```

# Exercise 1 - GPU Code (3D Cloth Scene)

- Cloth rendered as triangle strip
- IDX Buffer:

```
for (int k = 0; k < NUM_CLOTHS; k++) {
    for (int i = 0; i < GRIDY - 1; i++) {
        for (int j = 0; j < GRIDX; j++) {
            // CCW?
            indices.push_back(k * GRIDX * GRIDY + i * GRIDX + j);
            indices.push_back(k * GRIDX * GRIDY + (i + 1) * GRIDX + j);
        }
        // Primitive restart
        indices.push_back(-1);
    }
}

CD3D11_BUFFER_DESC iDesc(
```

- Jacobi iteration on GPU, 2 buffers, buf_in, buf_out, pingpong between them

# Exercise 1 - GPU Code (3D Cloth Scene)

```
    );
    for (auto i = 0; i < iters; i++) {
        srv_idx = !srv_idx;
        uav_idx = !uav_idx;
        if (integrator == MIDPOINT) {
            context->CSSetShaderResources(0, 1, &srvs[srv_idx]);
            context->CSSetUnorderedAccessViews(0, 1, &uavs[uav_idx], nullptr);
            context->Dispatch(GRIDX / NUM_THREADS_X, GRIDY / NUM_THREADS_Y, 1);
            context->CSSetShaderResources(0, 1, null_srv);
            context->CSSetUnorderedAccessViews(0, 1, null_uav, nullptr);
            simulation_cb.integrator = 3;
            context->UpdateSubresource(
                simulation_buffer.Get(),
                0,
                nullptr,
                &simulation_cb,
                0,
                0
            );
            srv_idx = !srv_idx;
            uav_idx = !uav_idx;
            context->CSSetShaderResources(0, 1, &srvs[srv_idx]);
            context->CSSetUnorderedAccessViews(0, 1, &uavs[uav_idx], nullptr);
            context->Dispatch(GRIDX / NUM_THREADS_X, GRIDY / NUM_THREADS_Y, 1);
            context->CSSetShaderResources(0, 1, null_srv);
            context->CSSetUnorderedAccessViews(0, 1, null_uav, nullptr);
        } else {
            context->CSSetShaderResources(0, 1, &srvs[srv_idx]);
            context->CSSetUnorderedAccessViews(0, 1, &uavs[uav_idx], nullptr);
            context->Dispatch(GRIDX / NUM_THREADS_X, GRIDY / NUM_THREADS_Y, 1);
            context->CSSetShaderResources(0, 1, null_srv);
            context->CSSetUnorderedAccessViews(0, 1, null_uav, nullptr);
        }
    }
    context->CopyResource(vertex_buffer.Get(), buffer_in.Get());
```

# Exercise 2 - Algorithm

```cpp
void RigidBodySystemSimulator::simulateTimestep(float time_step) {
     if (!running) {
        return;
     }
     num_run++;

    if (m_iTestCase == 3) {
        static std::mt19937 eng;
        static std::uniform_real_distribution<float> randomer(-1.5, 2);
        static int counter = 0;
        counter++;
        if (counter > 500) {
            add_box({ randomer(eng), 1.0, randomer(eng) }, { 0.3, 0.3, 0.3 }, 2);
            counter = 0;
        }
    }
```

# Exercise 2 - Algorithm

```
for (auto& rb : rigid_bodies) {
    if (!rb.movable)
        continue;
    rb.position += time_step * rb.linear_velocity;
    rb.linear_velocity += time_step * rb.force * rb.inv_mass;
    auto ang_vel = Quat(rb.angular_vel.x, rb.angular_vel.y, rb.angular_vel.z, 0);
    rb.orientation += time_step * 0.5f * rb.orientation * ang_vel;
    rb.orientation = rb.orientation.unit();
    rb.angular_momentum += time_step * rb.torque;
    auto inv_inertia = rb.get_transformed_inertia(rb.inv_inertia_0);
    rb.angular_vel = inv_inertia * rb.angular_momentum;
}
// Clear forces & torques
for (auto& rb : rigid_bodies) {
    rb.force = 0;
    rb.torque = 0;


}
if (m_iTestCase == 0) {
    auto& rb = rigid_bodies[0];
    cout << "Linear Velocity: " << rb.linear_velocity << "\n";
    cout << "Angular Velocity: " << rb.angular_vel << "\n";
    auto point_vel = rb.linear_velocity + cross(rb.angular_vel, Vec3(0.3, 0.5, 0.25));
    cout << "Velocity at point (0.3, 0.5, 0.25): " << point_vel << "\n";
    running = false;
}
handle_collisions();
```

Euler method

# Exercise 2 - Collisions

```cpp
void RigidBodySystemSimulator::handle_collisions() {
    if (rigid_bodies.size() < 2) {
        return;
    }
    bool resolve = false;
    Contact* ci = nullptr;
    CollisionData data;
    for (int i = 0; i < rigid_bodies.size() - 1; i++) {

        RigidBody& b1 = rigid_bodies[i];

        for (int j = i + 1; j < rigid_bodies.size(); j++) {

            RigidBody& b2 = rigid_bodies[j];
            // TODO: Vector is overkill here, fix
            std::vector<RigidBody*> pairs = { &b1,&b2 };
            std::sort(pairs.begin(), pairs.end(), [](RigidBody* a, RigidBody* b) {
                return a->type < b->type;
            });
            if (pairs[0]->type == RigidBodyType::CUBOID && pairs[1]->type == RigidBodyType::CUBOID) {

                ci = checkCollisionSAT(pairs[0]->obj_to_world(), pairs[1]->obj_to_world(), &data);
                resolve = ci && ci->is_valid;
                if (resolve) {
                    ci->bodies[0] = &b1;
                    ci->bodies[1] = &b2;
                    ci->cp_rel[0] = ci->collision_point - b1.position;
                    ci->cp_rel[1] = ci->collision_point - b2.position;
                }
            } else {
                int k = static_cast<int>(pairs[0]->type) | static_cast<int>(pairs[1]->type);
                auto collision_func = collision_map[k];
                Mat4 b1_world = pairs[0]->obj_to_world();
                ci = collision_func(pairs[0], pairs[1], b1_world, data);
                resolve = ci && ci->is_valid;
            }

            if (resolve) {
                // Apply position change
                resolve_positions(data);
                // Apply velocity change
                resolve_velocities(data, ci, pairs);
            }
            data.reset();
        }
    }
```

Collisions are mapped to correct collision function according to their shapes.
Cube - Cube is solved with SAT

```cpp
RigidBodySystemSimulator::RigidBodySystemSimulator() {
    m_iTestCase = 0;
    collision_map[5] = collision_box_plane;
    collision_map[6] = collision_sphere_plane;
    collision_map[2] = collision_sphere_sphere;
    collision_map[3] = collision_box_sphere;

}
```

# Exercise 2 - Collisions

```cpp
#include "RigidBody.h"
struct Contact {
    RigidBody* bodies[2] = { nullptr, nullptr };
    Vec3 cp_rel[2];
    int cp_idx;
    Vec3 normal;
    Vec3 collision_point;
    float relative_vel;
    float expected_vel = 0;
    float penetration;
    bool is_valid;
};

struct CollisionData {
    CollisionData() {
        contacts.resize(16);
        num_contacts = 0;
    }
    void reset() {
        contacts.resize(16);
        num_contacts = 0;
    }
    std::vector<Contact> contacts;
    int num_contacts;
};
```

Collision function returns Contact data which is then used to resolve positions and velocities.

```cpp
if (resolve) {
    // Apply position change
    resolve_positions(data);
    // Apply velocity change
    resolve_velocities(data, ci, pairs);
}
```

# Exercise 3 - Grid Structure

```
// Some documentation of grid for reference
// Flattened(default) version (indices). For EX in 2D:
/*
    0, grid_dim, 1, 1+grid_dim, ... grid_dim-1, 2*grid_dim -1,
    2*grid_dim 2*grid_dim + grid_dim...
*/
// For 3D:
/*
    0, grid_dim, 1, 1 + grid_dim, ... grid_dim - 1, 2 * grid_dim - 1
    grid_dim*grid_dim, grid_dim*grid_dim +1, grid_dim*grid_dim + grid_dim...
*/

// Example 4x4 block indices
/*
    0 2 4 6
    1 3 5 7
    8 10 12 14
    9 11 13 15
*/

// Example 5x5 block indices
/*
    0 2 4 6 8
    1 3 5 7 9
    10 12 14 16 18
    11 13 15 17 19
    20 21 22 23 24
*/
```

# Exercise 3 - Grid Initialization

```
// Set up initial values and boundary values
// We are using Drichlet conditions so we have
/*
    0 0 0 0 ... 0
    0 x x x ... 0
    0 x x x ... 0
    0 x x x ... 0
     ..
    0 0 0 0 ... 0
*/
```

# Exercise 3 - Explicit

```cpp
bool is_dim_odd = dim_y % 2;
// Stability conditions:
// In 1D:
// u_i^{t+1} = u_i^{t} + F * (u_i^{t} - 2 * u_i^{t} + u_{i-1}^{t} )
// In nD
// u_i^{t+1} = u_i^{t} + F * (u_ix^{t} - 2 * u_ix^{t} + u_ix{i-1}^{t} )
//                + F * (u_iy^ { t } - 2 * u_iy^ { t } + u_{ i - 1 }y^ {t})
                // ...
// Where F = alpha * time_step / dx2;
// Reorganizing we have
//  u_i^{t+1} = (1 - 2 * n * F) u_i^{t} + F (u_ix^{t} + u_{i-1}x^{t} ...)
// The stability condition is when 1 - 2 * n * F <= 0 => F <= 1/(2 * n)
// Note that larger F values cause current cell to have negative impact hence the solution blows up

// The limiting time_step value is
//  alpha * time_step / dx2 <= 1/(2*n)
// => time_step <= dx2 / (2 * n * alpha)
// With differing grid dimensions we have a * dt * (inv_dx2 + inv_dy2 + inv_dz2) <= 1 / 2
// => time_step <= inv_inv / (2 * alpha)
if (clamp_ts) {
    constexpr double EPS = 0.1;
    Real val = inv_inv * (1 - EPS) / (2 * (1 + EPS) * alpha);
    if (time_step >= val) {
        std::cout << "INFO: Instability detected, clamping time_step from " << time_step << " to ";
        time_step = val;
        std::cout << time_step << "\n";
    }
}
```

# Exercise 3 - Explicit

Basically for each axis, sum up the values of 2 neighbors and extract the value in the cell multiplied by 2. Multiply each value obtained from the obtained in the previous step with inverse distance between neighbors. Sum up all the values, multiply it with alpha and timestep to get the new value of the cell.

```
        //const Real F = alpha * time_step  * inv_d;

        new_grid_values[i] = (x_diff * inv_dx2 + y_diff * inv_dy2 + z_diff * inv_dz2) * alpha * time_step + it;
    }

    previous_grid->values = std::move(grid->values);
    grid->values = std::move(new_grid_values);

    return nullptr;
}
```

# Exercise 3 - Implicit (CPU)

Feed it to a solver

```
}
// perform solve
static SparsePCGSolver<Real> solver;
solver.set_solver_parameters(pcg_target_residual, pcg_max_iterations, 0.97, 0.25);
std::vector<Real> x(N, 0);
// preconditioners: 0 off, 1 diagonal, 2 incomplete cholesky
solver.solve(adaptive_step ? *A_local : *A, grid->values, x, ret_pcg_residual, ret_pcg_iterations, 2);
if (use_gpu) {
```

```
}

 previous_grid->values = std::move(grid->values);
 grid->values = std::move(x);
```

# Exercise 3 - Implicit (GPU)

```
// Fixed SparseMatrix representation
StructuredBuffer<float> x;
RWStructuredBuffer<float> x_out;
StructuredBuffer<int> row_start;
StructuredBuffer<int> col_index;
StructuredBuffer<float> mat_values;
StructuredBuffer<float> rhs;

cbuffer CB : register(b0) {
    int N;
}

[numthreads(32, 1, 1)]
void CS(uint3 id : SV_DispatchThreadID) {
    int row = id.x;
    if (row >= N) {
        return;
    }
    float sig = 0.0;
    float diag;
    for (int j = row_start[row]; j < row_start[row + 1]; j++) {
        if (row != col_index[j]) {
            sig += mat_values[j] * x[col_index[j]];
        } else {
            diag = mat_values[j];
        }
    }
    x_out[row] = (rhs[row] - sig) * 1 / diag;

}
```

Jakobi iteration on compute shader (Same sparse matrix representation, so have indirection)

Two buffers:

```
);
context->CSSetConstantBuffers(0, 1, diffusion_cb.GetAddressOf());
for (int i = 0; i < num_jacobi_iters; i++) {
    srv_idx ^= 1;
    uav_idx ^= 1;
    context->CSSetShaderResources(0, 1, &srvs[srv_idx]);
    context->CSSetUnorderedAccessViews(0, 1, &uavs[uav_idx], nullptr);
    context->Dispatch(TG_X, 1, 1);
    context->CSSetShaderResources(0, 1, null_srv);
    context->CSSetUnorderedAccessViews(0, 1, null_uav, nullptr);
}
```