GLSL Compute Shaders Mass-Spring-Damper simulation

Scaramelli Federico
Computer Science department
University of Milan
Milan, Italy
federico.scaramelli@studenti.unimi.it

Ivan Cimino
Computer Science department
University of Milan
Milan, Italy
ivanemanuele.cimino@studenti.unimi.it

I. Introduction

The project presented in this report aims to create a physical simulation based on a Mass-Spring-Damper system to reproduce various fabrics materials and ropes. The physics simulation has been written in GLSL language using its compute shader functionalities in order to exploit parallel computing offered by the GPU. Parallel computing has raised issues related to the fact that the constraints applied to this kind of simulations are usually solved sequentially following a specific order to avoid known problems.

To support the development, a dedicated engine has been created to allow a dynamic management of objects in the scene and possible future expansions of the project.

The results obtained were good both in terms of performance and of results relating to the simulation itself.

II. Engine Description

The engine developed tries to abstract all the main components necessary to handle and render the scene objects. Reusability is actually not very high due to the specific behavior of the mass spring simulation inserted in some of its classes. The main strength of the engine is its flexibility in adding or modifying features of the simulation itself.

Here there is a description of all the main component abstractions.

A. Main Loop and Rendering

The entry point is the *Application.cpp* where we can find the initialization and the main loop. Here we create the window, abstracted in the *Window.cpp* class. Before entering the main loop we have to add our starting objects to the scene.

The main loop calls the update method of the scene and of the *Renderer* object. The *Draw* function of the renderer component is invoked for each individual object by the scene update method, and simply calls OpenGL functions to draw on the screen the data contained in the buffers.

B. Game Objects

Game objects are abstracted with a class hierarchy. The main one has some pure virtual functions and defines some common behaviors for all kinds of game objects. Here is the structure of the hierarchy with a brief explanation of each component.

- *Transform component*
  Contains all the methods to handle the object into the space through transformations. It computes model and normal matrices starting from position, rotation and scaling data.
- *Mesh component*
  Contains vertices and indices data and has methods to send them to the buffers. It contains also the material component.
- *Material component*
  Contains all the parameters related to the material to be used in the fragment shader and specifies which fragment shader to use. It contains also the shader program component.
- *Shader Program component*
  It represents the shader program of the object, specifically the vertex and the fragment shader to be used. It provides useful methods to work with shaders such as the uniforms setting methods.
- *Buffers*
  An abstraction of both the *Vertex Array Object*, the *Vertex Buffer Object* and the *Index Buffer* have been implemented. To work with the VAO the *VertexBufferLayout* class was implemented. This class allows to define a structure layout for the vertices data contained in the VBO in order to be able to access them through strides and offset defined by the layout itself.

All classes derived by GameObject have their own creation and update methods to define custom behaviors. Another important component of the game objects is their UI component which can be customized deriving the *GameObjectUI* class.

C. Scene

The Scene class is probably the most important of the entire engine. It stores the lists of the various kinds of objects currently present into the game scene and contains all the update method calls into its own update method. It provides an interface to add and remove objects from the scene. It has been implemented as a singleton in order to be able to access them from everywhere into the code. This class is not really

generic, in fact it contains specific methods to handle the mass spring objects.

Fig. 1. The UI offered by the engine to handle the scene in real-time.

D. Light, Camera and Wind

Light, Camera and Wind objects are unique into the scene and handled by the *Scene* class separately from the other objects. They have their own creation and update custom behaviors, as well as a custom UI component.

E. Game Objects types

The main classes derived from the *GameObject* class are the follwing:

- *Primitive*
  Abstracts an object represented with a geometric primitive. It contains methods to initialize vertices and indices of cubes, spheres and cones.
- *Mass Spring*
  It is another virtual class from which are derived the specific classes of the mass spring, namely Cloth and Rope.
- *Colliding Sphere*
  Deriving from the Primitive class allows to handle spherical colliders interacting with the mass spring objects. The engine provides a custom UI to add, remove and handle them into the scene with a maximum amount of 10 colliding spheres at a time.
- *Light, Wind*
  Also these classes derive from the Primitive one in order to be able to better handle their instances into the scene through the visualization of a sphere and a cone gizmos, respectively.

F. Mass Spring objects

The two classes derived from the MassSpring one, namely Cloth and Rope, contain all the custom behavior of the two simulations, described in more details in the Simulation section. A system to handle presets was implemented for the Cloth class only, allowing to define different simulations to be easily instantiated into the scene.

III. Simulations

In this section are described the design choices related to the physics simulations.

A. Cloth

The cloth mass-spring system can be described as a grid of particles, each with its own mass. The total number of particles in determined by the number of rows and columns of the grid. The particles are connected to their neighbours through springs, which impress the elastic forces on each pair of linked particles. There are four types of springs, depending on which particles they connect.

- *Structural Springs*
  Springs between a particle and the neighbour particles in the up, bottom, left and right directions at a distance in the grid equals to one.
- *Sheering Springs*
  Springs between a particle and the neighbour particles in the up-left, up-right, bottom-left and bottom-right directions at a distance in the grid equals to one.
- *Bending Springs*
  Springs between a particle and the neighbour particles in the up, down, left and right direction at a distance in the grid equals to two. As the name suggests, these springs in particular act on the bending of the cloth. Depending on their strength the spots when the cloth curves can be more or less edgy.

For each of the simulated springs, we define various parameters in the MassSpring class, but there are also parameters unique to the cloth itself like the rest lengths of the springs, which depend on the spring type. The cloth also has a resolution, which is the desired number of particles by width and height.

B. Rope

The rope mass-spring can be described as a linear sequence of nodes, each with a fixed number of vertices around it. The rope simulations is simpler than the cloth one: the springs are all the same, the sequence of nodes is monodimensional rather than bidimensional, and there are fewer particles to simulate over time. Yet, in this simulation we have to recompute the positions of all the vertices of a node, which depend on the positions of next and previous nodes. For this reason, there are an extra buffer and shader for this simulation. Other than the rest length between nodes, the other parameters unique to this simulation are the radius of the rope and the weights used to calculate the vertices position around each node. Each weight is a couple of values used, together with the radius, as

a combination of coordinates for two axes around each node to establish the vertex position around the node.

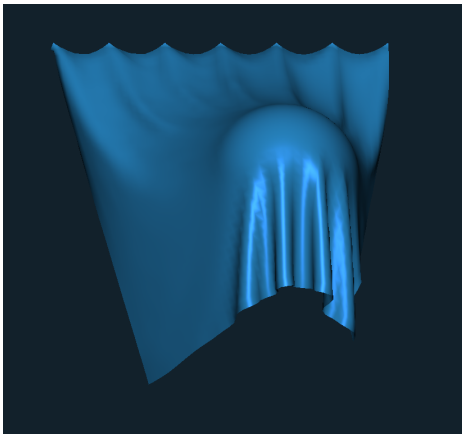

Fig. 2. A cloth using the preset *'curtain'* touching a colliding sphere.

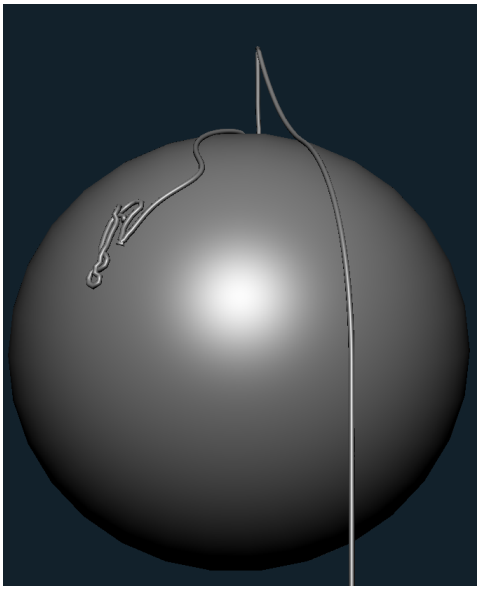

Fig. 3. A rope lying on a sphere with maximum friction.

IV. Compute shaders

Compute Shaders are a new type of shader introduced into the GLSL language starting from OpenGL 4.3. They allow to write GP-GPU code with a SIMD fashion in order to be able to exploit the parallelism offered by the GPU. Each compute shader is executed in parallel by a specific number of threads. The amount of threads to assign to the compute shader execution must be specified at the time of its dispatch. The organization of the threads takes place on two levels, through the construction of a three-dimensional grid of three-dimensional workgroups of threads. A workgroup is therefore a set of threads organized in three dimensions. Furthermore, a set of workgroups organized in three dimensions is assigned to a specific compute shader dispatch. GLSL provides derived input variables to obtain the index of the specific thread related to a single execution of the parallel computation. It is possible to obtain the local index referring to the single workgroup to which it belongs, as well as the global index referring to all the workgroups. The indices obtained are expressed in three dimensions and must be manually linearized, if necessary.

To organize the dispatch of the compute shader, a class dedicated to the abstraction of the shader program related to a compute shader has been implemented. The attributes of this class are *workGroupSize* and *workGroupNum*. While the *workGroupSize* value is specified manually, the *workGroupNum* value is calculated automatically based on the size of a single workgroup and the size of the data structure on which to perform the computation. This relationship is due to the fact that in a SIMD fashion approach, usually a mapping between the threads and the values of the data structure on which to perform the parallel computation is defined.

Listing 1. Cloth simulation compute shader creation

```
compShader.CreateProgram (
    { "clothSimulations.comp", ShaderType::COMPUTE }
);
compShader.SetWorkGroupSize ({ 16, 16, 1 });
compShader.SetWorkGroupNum ({ GetClothSize(), 1 });
```

As can be seen from *Listing 1*, a workgroup size equal to *16x16x1* has been specified for the compute shader related to the cloth simulation. This choice is due to the fact that the particles of the cloth are organized in a two-dimensional matrix. Thanks to this organization of the execution, within the shader it is possible to refer to a single particle using the thread index obtained through the variables provided by GLSL. It is therefore clear that each thread will be in charge of performing the calculations for the simulation of a single particle.

Listing 2. The first lines of the cloth simulation compute shader

```
uvec3 globalIdx = gl_GlobalInvocationID;

// Linear index of the particle
uint linearIdx = globalIdx.x
               + (globalIdx.y * clothDims.x);

// Safety check
if (globalIdx.x >= clothDims.x
    || globalIdx.y >= clothDims.y)
            return;

// Keep pinned particles fixed
if (IsPinned(linearIdx)) {
        verticesOut[linearIdx].pos
                = verticesIn[linearIdx].pos;
        verticesOut[linearIdx].oldPos
                = verticesIn[linearIdx].oldPos;
        verticesOut[linearIdx].vel
                = vec4(0.0);
        return;
}

vec3 position = verticesIn[linearIdx].pos.xyz;
vec3 oldPosition = verticesIn[linearIdx].oldPos.xyz;
vec3 velocity = verticesIn[linearIdx].vel.xyz;
```

The safety check is useful to avoid the execution of threads which have not a direct mapping with a particle of the cloth.

It's a common pattern in GP-GPU algorithms as sometimes it's not possible to use a workgroup size and amount that perfectly fit the data size.

Specifically, relatively to the mass spring simulations described in this report, each compute shader is designed to be dispatched using workgroups composed by a total of 16 or 32 threads. A total of three compute shaders have been implemented to handle the physics simulations, the application of positional constraints and the update of vertices positions in the rope simulation, respectively.

Each simulation has a main buffer and a secondary buffer, which are swapped after each shader execution. This is due do the fact that each thread needs to read data of multiple particles while other threads are writing those data in the same buffer. This would lead the program to race conditions. After each swap the input buffer becomes the output one, and vice versa. The only exception is the vertex buffer in the rope simulation, since each invocation reads and updates only its own data from it.

After each compute shader dispatch, barriers are put in order to wait for all threads to finish their work. Every shader receives the necessary parameters as uniforms, updated if they change during the scene update.

Listing 3. Compute shaders dispatch and buffers swap

```
simulationStageComputeShader.Use();
BindComputeBuffers (0, 1);

simulationStageComputeShader.Compute();
simulationStageComputeShader.Wait();

constraintsStageComputeShader.Use();
BindComputeBuffers (1, 0);

constraintsStageComputeShader.Compute();
constraintsStageComputeShader.Wait();
```

In the *Listing 3* is possible to identify the use of three methods offered by the class which abstract the compute shader program. The *Use()* method tells OpenGL that this specific shader program is going to be used. The *Compute()* method execute the dispatch of the compute shader with the specified workgroup sizes. The *Wait()* method tells OpenGL to wait until the previously dispatched compute shader has terminated its execution. The barrier has the *GL_SHADER_STORAGE_BARRIER_BIT* bit which activate the barrier on memory operations about storage buffers, since compute shaders write and read from storage buffers.

A. Physics shader

The physics shader is the compute shader that compute the evolution of the simulation. At the start of the shader the total force applied to a particle is computed.
The considered forces are:

- *Gravity force*
  Equal for all particles.
- *Elastic forces*
  Calculated simulating springs connecting the particle with the neighbouring particles. In total each particle has 12 springs, and for each spring the elastic force is computed through the Hooke's Law.
- *Wind force*
  This force can be zero, depending on the direction and the position of the wind with respect to the particle. If present, this force is weighted with the angle between the wind direction and the cloth particles in a cone centered in the wind position.

Listing 4. Elastic force computation

```
vec3 vectorToOther = otherPosition - myPosition;
float distance = length(vectorToOther);

// Hooke's law
vec3 force = elasticStiffness
           * (distance - restLength)
           * normalize(vectorToOther);
```

Listing 5. Wind force computation

```
vec3 vectorToVertex = position - wind.position.xyz;
vec3 windDir = wind.forward.xyz;
float dotWindZ = dot (windDir,
                      normalize(vectorToVertex));
float windMult = wind.forceMult;
vec3 maxRadiusPoint = vec3( 0,
                            wind.fullForceRadius,
                            0)
                    + windDir
                    * wind.referenceDistance;
float maxDot = dot( windDir,
                    normalize(maxRadiusPoint));
vec3 attRadiusPoint = vec3( 0,
                            wind.fullForceRadius
                            + wind.attRadius,
                            0)
                    + windDir
                    * wind.referenceDistance;
float attDot = dot( windDir,
                    normalize(attRadiusPoint));
if (dotWindZ < attDot) {
        windMult = 0;
}
if(dotWindZ < maxDot) {
        float ratio = (dotWindZ - attDot)
                    / (maxDot - attDot);

        windMult = windMult * ratio;
}

return windDir * windMult;
```

After the total force is being computed, the drag simulating the air resistance is applied, following the Stokes' Law.

Listing 6. Stokes' Law

```
// Stokes's law
totalForce -= velocity * drag;
```

Following, after the acceleration is computed, the velocity and position of the particles are updated using the Symplectic Euler integration method.

Listing 7. Symplectic euler integrator

```
vec3 newVelocity = velocity
                 + acceleration * deltaTime;
vec3 newPosition = position
                 + newVelocity * deltaTime;
```

Finally the new values are written in the output buffer.

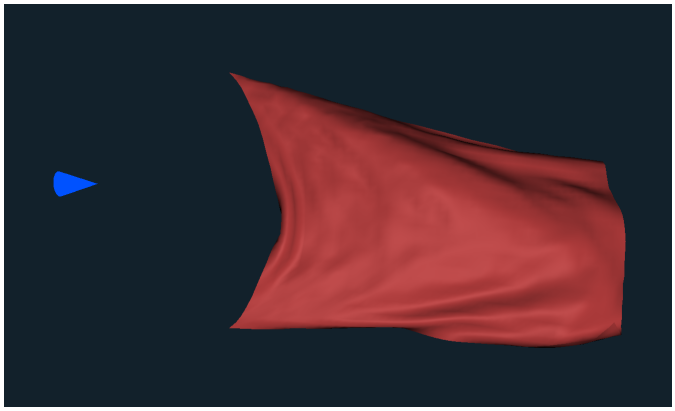

Fig. 4. A cloth using the preset *'flag'*. The blue cone is the wind gizmo.

B. Constraint shader

The constraint shader is the compute shader that process both the collisions and the positional constraints to be applied to the particles after the new position and velocity are computed. The only positional constraint applied states that the springs cannot elongate more than a certain percentage from the rest length. For each spring, the correction to respect this constraint is computed and then applied to the position of the particle.

Listing 8. Positional constraints computation

```
vec3 correction = vec3(0,0,0);
float diff = distanceToOther - maxDistance;

if (diff > 0)
{
        correction = normalize(vectorToOther)
                     * diff * 0.5;

        correction = (IsPinned(otherIndex)) ?
                correction * 2 :
                correction;
}

return correction;
```

After this constraints application, two kind of collisions are calculated: self-collisions and collisions with the colliding spheres present in the scene.

- *Self collisions*
  Collision between particles of the cloth. They are computed calculating the distance between each particle of the cloth and the particle that the compute shader invocation is in charge of. If the distance is less than a certain threshold, depending on the rest length, then a correction is computed to bring the particle away from the ones it's colliding with. After the final correction is computed, it is applied to the particle position.
- *Spheres collisions*
  Collisions between the particle and spheres present in the scene. If the distance between the particle and the sphere is less than the radius of the sphere then the position is corrected putting the particle at the right distance, along the direction between the center of the sphere and the particle.

Listing 9. Self-collisions computation

```
for (int i = 0; i < size; i++)
{
  vec3 vectorToOther = verticesIn[i].pos.xyz
                     - verticesIn[linearIdx]
                       .pos.xyz;
  float dist = length(vectorToOther);
  vec3 dirToOther = normalize(vectorToOther);

  float correctionMult;
  vec3 currentCorrection = vec3(0,0,0);

  correctionMult = (IsPinned(i))
                 ?  1
                 :  0.5;

  float diff = dist - radius;
  currentCorrection = (diff < 0
                      && i != linearIdx)
                      ? currentCorrection
                        + dirToOther
                        * diff
                        * correctionMult
                      : currentCorrection;

        correction += currentCorrection;
}
```

Listing 10. Particle-Sphere collisions computation

```
vec3 sphereDist = correctedPosition
                - spheres[i].sphereCenter.xyz;

if (length(sphereDist) < spheres[i].sphereRadius
                       + restLenHV
                       * sphereDist)
{
  // If the particle is inside the sphere,
  // push it to the outer radius
  correctedPos = spheres[i].sphereCenter.xyz
               + normalize(sphereDist)
               * (spheres[i].sphereRadius
               + restLenHV
               * sphereDist);
  // Damp velocity with sphere friction
  correctedVel *= sphereFriction;
}
```

After all corrections have been applied the velocity of the particle is updated, and then also the normal of the particle is updated through the *normal smoothing* technique. To do so, the cross product between each pair of edges of the particle neighbours is computed. The final normal of the particles is the average of all the computed cross product weighted using the angle between the two edges.

C. Vertices update shader (rope only)

Rope is threated a bit differently than the cloth. The physical simulation act on nodes, updating their position and velocity. The vertices position is then updated with the nodes since each node has eight vertices around it. Instead of simply move each vertex by the same amount the relative node has moved the vertices position is updated considering also the next and the previous nodes. This is done in order to make the visual aspect of the rope less edgy in the internal nodes. Each invocation

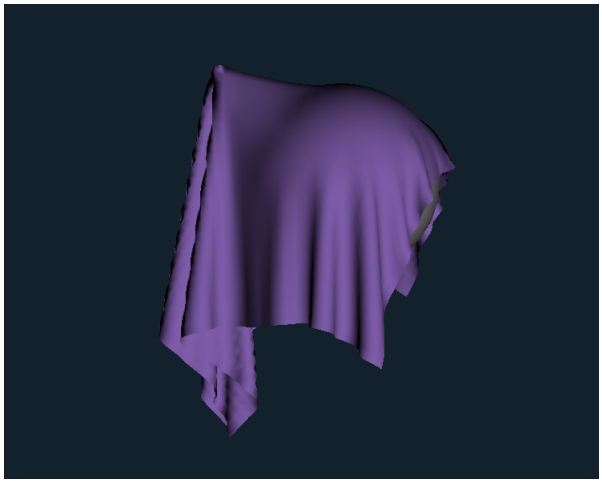

Fig. 5. A cloth using the preset *'towel'*. Here it's possible to observe self-collisions.

calculates the position of a vertex using a set of two axes, a combination of weights and the radius. To calculate each axis, the direction from the current towards the next node and from the previous to the current node are taken, adding and normalizing them to get a forward axis. Then, to get the sideward axis the normalized cross product between the forward axis and a global up direction is computed. If the two axes are colinear, the sideward axis is obtained rotating the forward axis by 90 degrees clockwise. Finally, the upward axis is calculated from the normalized cross product between the forward and sideward axes. The sideward and upward axes are then used to calculate the vertex position using the weights and the radius applied to the axes.

Listing 11. Rope vertices computation

```
// Calculate the vertex position around the normal
// using an orthonormal base calculated from the
// direction towards the next / previous nodes
vec3 forwardNext = normalize
                   (nextNodePosition - nodePosition);
vec3 forwardPrec = -normalize
                   (precNodePosition - nodePosition);

// Forward used is the average of the
// direction towards next node and precedent node
vec3 forwardFinal = normalize
                    (forwardNext + forwardPrec);

vec3 origin = nodesIn[nodeIndex].pos.xyz;;

float offsetX = vertexWeigths[wheightsIndex].x
                * radius;
float offsetY = vertexWeigths[wheightsIndex].y
                * radius;

vec3 rightFinal = normalize(Rotate(forwardFinal));
vec3 upFinal = normalize(
                    cross(forwardFinal, rightFinal)
               );

vec3 positionRight = rightFinal * offsetX;
vec3 positionUp = upFinal * offsetY;

vec3 vertexPosition = positionRight + positionUp;
vertexPosition += origin;
```

V. Performance Analysis

We presented a parallel approach to implement a mass-spring simulation exploiting the capabilities of GLSL compute shaders. Let's analyze the achieved performances, the optimizations applied, the current bottlenecks, and some possible improvements.

All the tests presented here has been executed with this setup:

- CPU: AMD Ryzen 5600x 3.7GHz
- GPU: NVIDIA GeForce RTX 3070 8GB
- RAM: 16GB 3600MHz
- RESOLUTION: *1920x1080*

Simulation is set to execute 16 simulation sub-steps with a timestep and a fixed delta time equal to 0.016, that is a 60 fps simulation.

A. Cloth

The first use case taken in analysis is about the *curtain* simulation with a size of 50x50 particles.

With all the features enabled the average fps value is around 1700 fps.

Just by adding a colliding sphere into the scene the performances drop to 1600 fps. Reaching the maximum amount of spheres, equals to 10, performances drop to 1400 fps. These firsts observations led us to the conclusion that sphere collisions have an impact related to the fact that we compute an intersection test for each particle and each sphere on each sub-step of the simulation. This means, in this particular case, that we compute 400.000 intersection tests for each simulation step, that is 1/60 of a second.

Passing to the wind performance analysis, it does not led us to any fps drop by changing its parameters. This is because we execute the same exact computations with any settings. By totally disabling it, we didn't registered any performance improvement.

The heaviest part of the simulations is related to self-collisions. By disabling them we reach an fps amount equal to 2600, that is an improvement of 52,94%. Self-collisions are so impactful because we compute particle-to-particle a intersection test for each couple of particles on each sub-step of the simulation. This means we compute 6.250.000 tests on each sub-step, equals to 100.000.000 tests during 1/60 of a second. Even if we compute tests related to a single particle on a dedicated GPU thread executing all the particle threads in parallel the computation remains really expensive as proved by this test.

Others cloth presets led us to similar performances.

B. Rope

Rope simulation is less computationally expensive, mainly because we have less particles to compute. In fact, a rope with 256 particles already provides a good simulation result. With all the features enabled we registered 3600 fps.

An unexpected result is about self collisions. In fact, by disabling them on the rope simulation, we don't register any performance improvement. This is certainly due to the amount of tests performed during a single simulation sub-step, that in

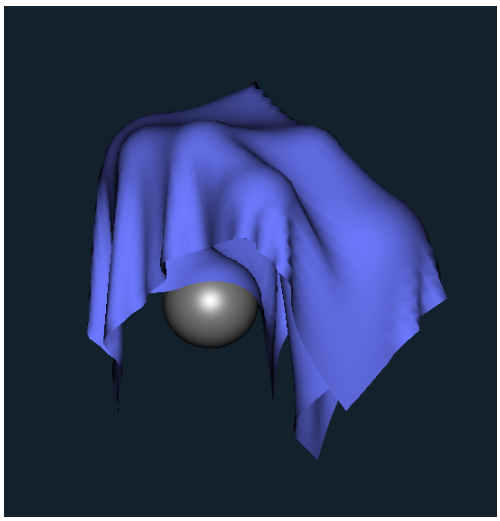

Fig. 6. A cloth using the preset *'towel'* with an higher resolution (*90x90*) colliding with 9 spheres.

this case are 1.048.576, about 1/6 of those required by the cloth taken in analysis. To prove it we performed a test with a curtain composed of the same amount of particles and we registered exactly the same performances.

Adding 10 spheres, instead, led the simulation to a drop of 16,66% of the total fps, reaching an amount of 3000 fps.

C. Optimizations

To optimize the parallel computation by fully exploiting the hardware acceleration provided by the GPU we apply divergence elimination techniques. GPU threads are well optimized only if they compute the same instruction at the same time in parallel. This is due to the SIMD approach of the GPU execution model.

To reduce divergence we used branching as less as possible, substituting *if statements* with ternary operators or boolean multiplications.

Another optimization is related to the amount of particles. In fact, setting them equal to a multiple of 16 we can exploit all the potential of parallel computing, leaving no thread without work. This is due to the fact that we launch the compute shaders with workgroups of 256 threads.

D. Bottlenecks

As shown by performance analysis the main bottlenecks are related to collisions, especially the self-collisions. Even if we execute all this computations in parallel, the amount of computing power needed is too much and it would been needed to exploit some acceleration algorithms in order to improve the performances.

E. Possible Improvements

As mentioned on the previous point, acceleration data structures and algorithms would been needed to improve performances anymore. Exploiting local BVH at mesh level we could reduce the amount of intersection tests performed, providing a certain improvement on performances.

VI. Conclusions

In conclusion it's possible to affirm that the performances obtained through the parallel implementation of a mass-spring system through the use of compute shaders in GLSL are good despite the many applicable improvements. The fidelity obtained from the physical simulation is equally good and, although the parallel application of the constraints is more problematic than the sequential one, it was still possible to implement a stable simulation even if placed in stressful situations. For future implementations, as anticipated, it would be possible to improve performances through the use of acceleration data structures and related algorithms.