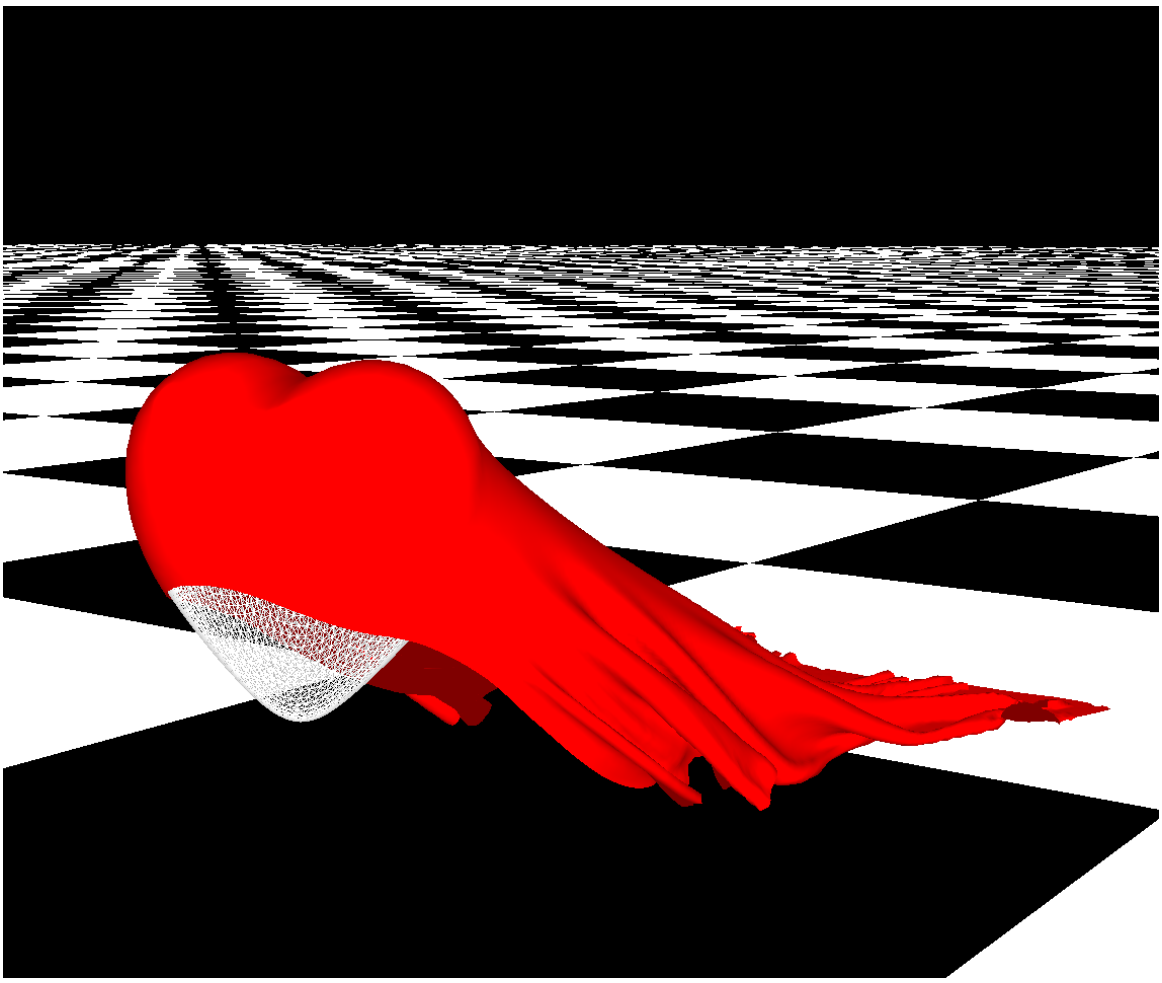
Final project report

Ensimag 3A MMIS [EXTERNAL CANDIDATE]

Courses: GP-GPU & Computational physics



Cloth simulation on GPU

(Valentine's Day themed)

Hélène HASSAN

February 16, 2024

Project link: `https://github.com/helenhsn/cloth-simulation`

Contents

I Introduction

As I stand as an external candidate for the semester and therefore am not required to participate in the GP-GPU class (nor the computational mechanics class), this project was more of a personal project. That is why I decided to choose a subject that was first and foremost fun and challenging. The keen interest I hold for computer graphics as well as the contents of the GP-GPU and computational physics classes pushed me to choose a topic related to this field, at the limit between computer science, physics and mathematics: a cloth simulation.

Clothes belong to a specific type of physics objects called *deformable bodies*, which includes many more real-life examples that you may be familiar with: liquids (viscous or not), hair, jelly bodies (like gummy-bears), and so on. In computer graphics, visually realistic physical simulations of deformable bodies are called *soft bodies*, as opposed to rigid bodies which does not possess the property to deform themselves over time and are exposed to different type of external and internal forces.

Any physically based simulation requires several steps in order to work, each of them having its own purpose. When approaching such a topic, one needs to understand that every choice made regarding the simulation whether it be the choice of data structures to store the mesh, the physical model of the cloth, of the physics dynamics will undeniably have an impact later on the improvements that can be made over time as well as the general structure of the code.

The basic features of the simulation I had in mind when submitting my topic last October were the following:

→ **The cloth can collide with any mesh.**

→ **The cloth can react to the following external forces: wind, gravity, air friction, kinetic/static friction(s) due to the contact with another mesh.**

→ (*Optional*) **The cloth can collide with itself.**

→ **The simulation is done on a framework made from scratch using C++/Cuda/OpenGL.**

→ **The simulation runs entirely on GPU, from the physics steps to the rendering step.** No transfer to CPU is allowed, so this means using Cuda/OpenGL interoperability is mandatory.

→ **The simulation runs in real-time (framerate $\geq$ 30fps)**

Even though the first three elements of the features are less related to parallel programming, they were the ones that were the more time consuming, given that I had to read a lot of research papers to figure out what my options were for each feature and how to build the solution around my choice for each of them. Moreover, the computational physics course taught notions mainly related to rigid body physics or CPU-solutions which were not feasible for me.

In the end, the only feature I have not had time to implement properly is the optional one which involves more thinking, even though I did make some advancements regarding that. The conclusion section will dive deeper in that topic which includes GP-GPU notions that I wish I had time to do in order to show them on this report.

II Research and methodology

For this section, only a summary of the actual research I did is presented. All sources are available at the end of the report, in the Sources section.

A Cloth Model

The model of the cloth determines its *internal forces* (also referred as *internal dynamics*) which are, by definition, independent from the environment in which it evolves.

There are many type of cloth models that can be considered which are obviously varying regarding the degree of realism and difficulty of implementation they offer. For example, I chose to model a cloth as set of vertices linked between them (ie, a *mesh*) which means that I am mainly going to be interested in the dynamics of the mesh, but some advanced papers consider cloth in at a smaller level, for example by considering the *yarns* constituting the cloth.

However, it stands to logic that the more realistic elements the model takes into account, the more difficult it will be to implement it. Indeed, a cloth model based on a simple squared mesh can quickly become quite difficult to manage, even when doing so on a GPU.

For instance, some models (called *continuous models* in computational physics) consider the cloth as a single body, which means that each triangle of the mesh is not filled with void but is instead treated as a whole component. The majority (if not all) of these models are energy-based models, which means that the internal dynamics of the cloth revolve around the expression of the potential energy of each force applied on each triangle. On the opposite side, *discrete* models aim at simplifying the model by considering the cloth as a simple set of vertices linked by *springs*. The latter models are simpler to approach, which is the case of the famous mass-spring model which I am going to describe briefly.

A - a Mass-spring model

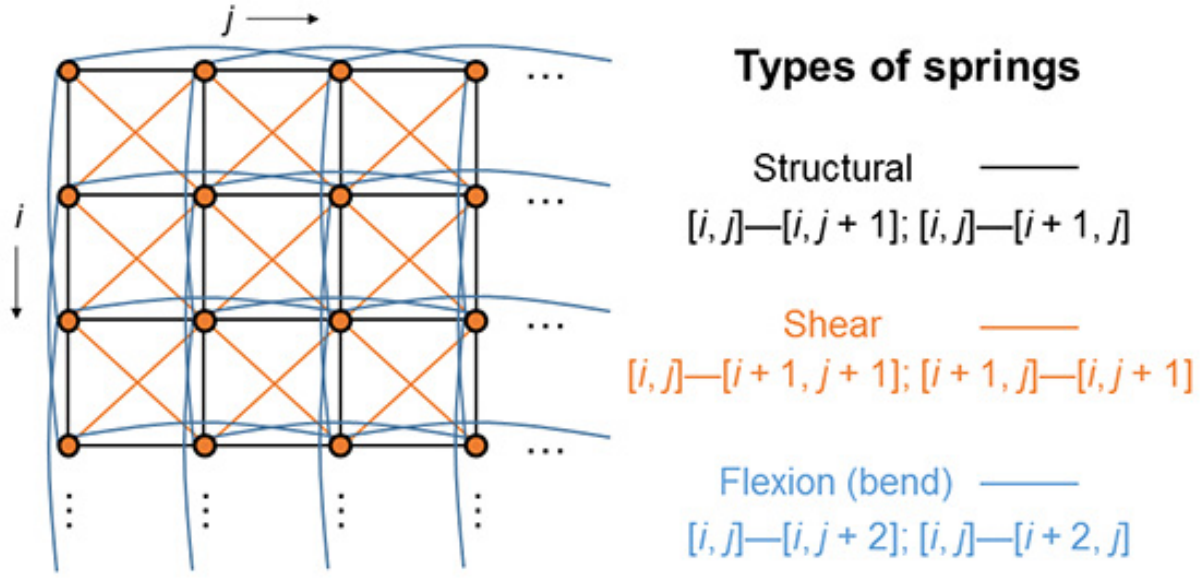


Figure 1: Spring-mass scheme used for the cloth model. // Source: https://ics.uci.edu/ shz/courses/cs114/docs/proj3/index.html

The mass-spring model is pretty simple to understand: each vertex is represented as a mass that is linked to at most 16 other vertices (masses) in the mesh. Each link is represented by a physical spring that possesses stiffness and damping coefficients. As we can see from the above scheme, there are 3 types of springs which depends on the distance between the adjacent vertices. The purpose of introducing these types of spring is to mimic different behavior of the cloth.

The mass-spring is very famous as it has the advantage to be implemented quite easily both on CPU and GPU compared to the other models. Still, this model just serves as an illusion given that the mesh does not behave as a whole and does not depend on real physics parameters.

Once the internal forces are defined, we can move on to the next step which is more practical: the integration, which aim for updating the physics metrics (position, velocity...) of each vertex using a numerical scheme.

B Integration step

B - a Numerical scheme choice

There are two main of numerical schemes: *explicit* and *implicit*. Explicit schemes such as the forward/backward Euler methods, the Verlet method, the middle point (or Runge Kutta 2) method or even Runge Kutta 4 method are usually known to be less stable than implicit schemes.

However, the use of an implicit scheme in the case of the cloth simulation always involves finding and computing the inverse of a sparse matrix which can be done on GPU using a modified conjugate gradient method. This algorithm can constitute an entire project by itself as there are many elements

to implement and optimize for it to work properly.

As this is a personal project, I started by implementing the Verlet integration following by RK4 integration which is known to be more stable with mass-spring based models. The downside of this scheme is that it is requires more computation: both the internal and external dynamics of the cloth have to be updated 4 times per frame in order to update the position and velocities of the vertices.

B - b Structural choices regarding the code

Mesh class

Based on the mass-spring model and on the explicit RK4 scheme, I created a ***Mesh*** class based on vertices, normals, texture coordinates (if any) and indices inputs. The constructor of this class initializes OpenGL buffers that are shared between Cuda and OpenGL using OpenGL/Cuda interoperability. In practice the buffers are required by Cuda in the first place in order to perform the physics computations and are then handed down to OpenGL which can perform the rendering.

This sharing of the position, normals and indices buffers allows computation to remain only on GPU which is a good thing for the framerate as transfers between host (CPU) and device (GPU) (especially from GPU to CPU) can be quite expensive when dealing with large arrays.

Every object in the scene inherits from this ***Mesh*** class (for example the ***Plane***, ***Sphere*** classes or even classes that represent meshes that are imported from .obj or .ply files, respectively named ***MeshFromOBJ*** and ***MeshFromPLY***)

Explicit Solver class

The ***ExplicitSolver*** class handles all computations required by the RK4 scheme. In order for the computations to be ran efficiently, one needs to be careful as how the buffers are initialized for the scheme update.

Buffers used for the integration step

→ Every buffer related to the position, velocity, acceleration and normal of the vertices are stored simply using the rule **1 element in the buffer = 1 vertex = (float, float, float)** . I must point out that the acceleration buffer corresponds in reality to the sum of all forces on each vertex but it makes no difference as every vertex has the same mass.

→ Every buffer related to the internal forces (spring) is initialized using the rule: **1 element in the buffer = 1 spring = 1 couple of vertices IDs = (int, int)**. Storing internal forces that way allows quicker computations when updating the internal dynamics of the cloth, which is something that we need given that RK4 scheme runs this step 4 times per frame.

$$\begin{cases} k_1 &= f(t+h, y+h) \\ k_2 &= f(t+h/2, y+hk_1/2) \\ k_3 &= f(t+h/2, y+hk_2/2) \\ k_4 &= f(t+h, y+hk_3) \\ y &= y + \dfrac{h}{6}(k_1+2k_3+2k_3+k_4) \end{cases}$$

Here, for the cloth model leads us to define $y = \begin{bmatrix} \dot{x} \\ x \end{bmatrix}$ where $\dot{x}$ and x are respectively the velocity and position. The function f that depends on the time-step h (first parameter) is responsible for computing the set of internal and external forces based on the provided velocity and position buffers (second parameter). These buffers are equal to that of the previous frame for k_1, and then an offset is added for the other k_i factors (with $i \geq 2$). After computing and updating the internal and external forces buffers for the four components k_i, the update of the position and velocity buffers can be translated as the following for the case of our cloth:

$$\begin{cases} x &= x + \dfrac{h}{6}(v_1+2v_3+2v_3+v_4) \\ v &= v + \dfrac{h}{6m}((F_{i1}+F_{e1}) + 2(F_{i2}+F_{e2}) + 2(F_{i3}+F_{e3}) + (F_{i4}+F_{e4})) \end{cases}$$

Where m is the mass of each vertex (the same for all vertices in the mesh), F_i corresponds to the sum of the springs forces for each vertex and F_e corresponds to the sum of the external forces (wind, air friction and gravity) for each vertex. I must point out that the gravity is computed once for the whole simulation but the air friction and wind depends respectively on the velocity and normal of each vertex and therefore need to be computed each time those buffers change.

Integration step algorithm

Let N_{cloth} be the number of vertices on the cloth's side so that the total number of vertices of the cloth mesh is N_{cloth}^2. As well as for the cloth, the spring buffers are also treated as 2D arrays, in practice I found that doing this makes the execution of the kernel that updates the internal forces faster. For every kernel call, blocks of size (32,32) are preferred.

The overall step can be described as followed:

1. Update of internal and external forces (k_i computation) for $i \in [\![1, 4]\!]$:

a) UPDATE_BUFFERS_SCHEME

– Grid Dim=$((N_{cloth})/32, (N_{cloth}+31)/32, 1)$ – Block Dim=$(32, 32, 1)$.

For each type of spring (shear/bend/stretch) do :

· ***b) UPDATE_INTERNAL_FORCES***

– Grid Dim = $((sqrt(N_{spring})+31)/32, (sqrt(N_{spring})+31)/32, 1)$ – Block Dim = $(32, 32, 1)$.

· CUDA_DEVICE_SYNCHRONIZE

c) UPDATE_EXTERNAL_FORCES
CUDA_DEVICE_SYNCHRONIZE

– Grid Dim=$((N_{cloth}+31)/32,(N_{cloth}+31)/32,1)$ – Block Dim=$(32,32,1)$.

2. Update of the vertices' position and velocity :

a) UPDATE_SCHEME
CUDA_DEVICE_SYNCHRONIZE

– Grid Dim=$((N_{cloth})/32,(N_{cloth}+31)/32,1)$ – Block Dim=$(32,32,1)$.

B - c Zoom on the update of the internal forces

The code corresponding to phases **1)a)**, **1)b)** and **2)a)** is pretty straightforward as it really does what a sequential algorithm would do with a for loop as each thread is updating each buffer element independently from one another. Loading each row in shared memory could have been an option to make some improvements for these kernels but since they do not have a high impact on performance, I decided to focus on the internal forces kernel and the future collision step.

The ***UPDATE_INTERNAL_FORCES*** is slighlty different from the other kernels as it deals with other type of data than the usual buffers (position, velocity, normals...). Here is the code for this kernel:

```
int j = threadIdx.x + blockIdx.x*blockDim.x;
int i = threadIdx.y + blockIdx.y*blockDim.y;
int tid = j*N+i;

if (tid < maxTid )
{
    glm::ivec2 ids = springIds[tid];
    glm::vec3 fSpring = computeSpringForces(isStretchSpring, ids.x, ids.y, xIter[ids.x], vIter[ids.x], xIter[ids.y], vIter[ids.y], L, Ks, Kd);

    float *addA = &FIter[ids.x].x;
    float *addB = &FIter[ids.y].x;

    int seed = ids.x*ids.y;
    glm::ivec3 offset = glm::ivec3(seed, seed+1, seed+2) % 3;

    atomicAdd(addA, fSpring, offset, seed);
    atomicAdd(addB, -fSpring, offset, seed);
```

Figure 2: Screenshot of the source code for the ***UPDATE_INTERNAL_FORCES*** kernel.

Overall, the kernel performs 6 loads from global memory and uses a custom atomicAdd (using vec3=float3 data instead of just float/int) to update the forces buffer of each vertex involved in the spring. Indeed, the use of atomic operation is here necessary since several threads might be accessing the same address in memory: this is due to the fact that each vertex in the mesh can be linked to at most 4 other vertices for each type of spring, so a vertex ID can be present in at most 4 elements of the spring buffer (for each type of spring).

One way to counter this problem has been to add a pseudo-random offset to the addresses inside each float-atomicAdd so that even if the same thread access to the vec3-atomicAdd at the same time, they would still be accessing the addresses of the x,y and z components with some delay. The seed for the pseudo-random offset is based on the ID of the two vertices in the spring as we can see in the kernel code above. The code for the custom atomicAdd is the following:

```
__device__ void atomicAdd(float *addr, glm::vec3 val, glm::ivec3 offset, int seed)
{
    atomicAdd(addr + offset.x, val[offset.x]);
    atomicAdd(addr + offset.y, val[offset.y]);
    atomicAdd(addr + offset.z, val[offset.z]);
}
```

Figure 3: Screenshot of the source code for the vec3-atomicAdd which also includes pseudo-randomization for the access of x,y and z components.

One way to counter this problem would be to build a buffer for each specific type of spring, for instance one buffer for the structural spring that goes from the current vertex to the bottom, one other buffer for the one that goes from the vertex to the left, etc. This solution would have taken the same amount of memory (as we still store each spring once) but more large arrays would have had to be defined on the device. Also, more kernel calls should have been made.

C Collision step (Cloth/Object collisions)

This step is usually the bottleneck of any physically-based simulation, even more when every type of collider is considered. Most of the cloth simulations we can observe on internet (Youtube or Github) often use objects that can be generated using operations (intersection, union) on signed distance field which can be as simple as a sphere or a cube. Such objects are part of what is called *constructive solid geometry* in computer graphics. However, limiting collisions to this type of objects is not really fun and many research papers approach the subject of collisions for any type of mesh, which is a good starting point.

Collision algorithms usually include two methods: a collision detection and a collision response. The collision detection can be either be *discrete*, meaning that the algorithm will search for intersection points at each time-step, or *continuous* where it searches for the first time of intersection between the two objects. Continuous detection algorithms are much more costly but have the advantage to be pretty accurate compared to discrete methods. One of the main problem of discrete collisions is as it depends on the time-step chosen for the simulation. The higher the time-step, the more likely the algorithm will miss collisions and lead to interpenetration cases between the colliders, so again there is always a trade-off when choosing one method over another.

As the majority of research papers approaching physics simulation on GPU are mixing the two methods, I decided to go first for the discrete collisions.

As I had some experience with ray-tracing algorithms before starting this project, I decided to use that at my advantage. The idea here is to consider each vertex of the mesh as a source of ray pointing in the direction of the vertex (and pointing in the other way, as objects can come from both directions). Each ray is then treated in parallel inside a kernel and tested against a Bounding

Volume Hierarchy (BVH).

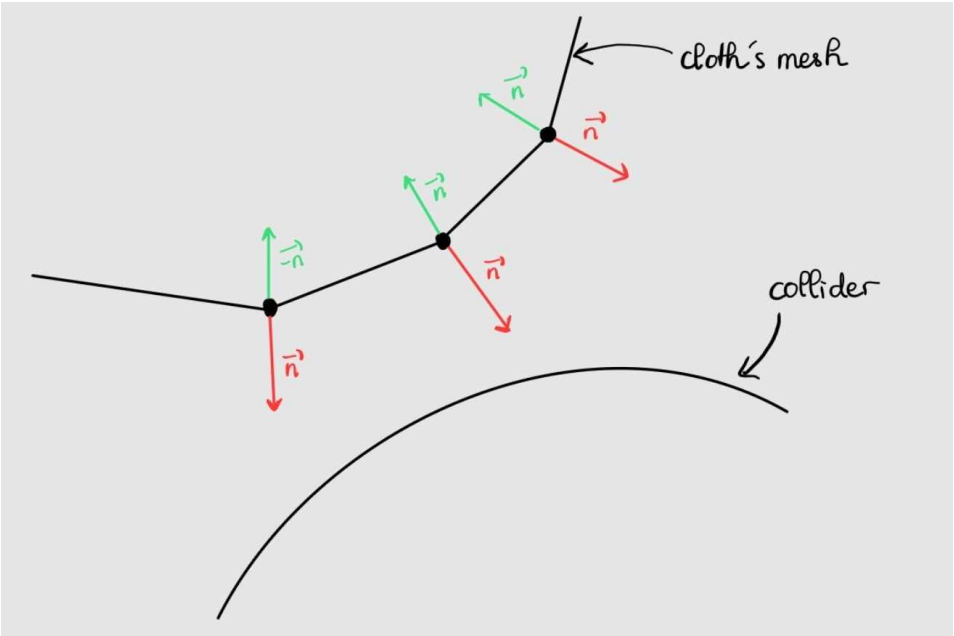


Figure 4: Scheme illustrating the collision detection

As seen above, the direction I used to cast the rays is the normal of each vertex (and its opposite vector). One might think that the velocity is a more appropriate vector to cast the rays as it corresponds to the movement of a vertex. In practice using the velocity vector was not a great idea as it can be almost canceled at certain times of the situation, which makes the raytracing algorithm invalid, for instance when the cloth's vertex just began to intersect the collider.

Initialization of the BVH

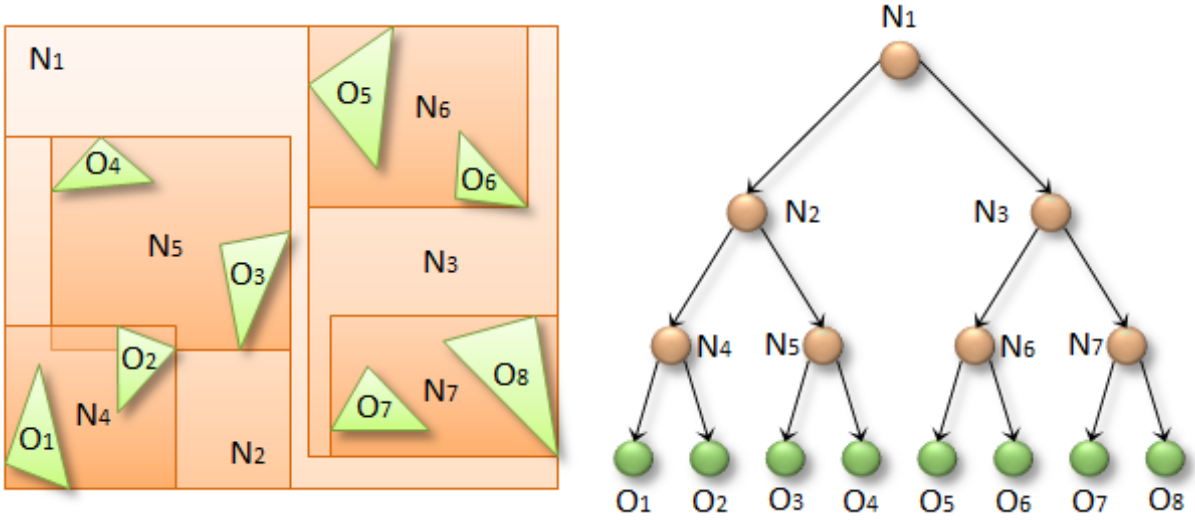


Figure 5: Scheme of a basic BVH // Source: NVIDIA Technical Blog, article by Tero Karras

A BVH is a type of tree where each node is associated with a subset of primitives (here: triangles) of the given 3D mesh. Each internal node has a bounding volume that encompasses the set of primitives it contains. This hierarchy helps build an organization between the different geometry parts of the mesh so that when searching for intersections with another object, simpler tests can be executed to discard all nodes that are to far from this other object. This way, unnecessary tests (primitive tests, often more costly than simple bounding volume tests) are avoided.

In the case of this project, the bounding volume chosen is the Axis-Aligned Bounding Box (AABB). Also, each leaf node can have up to 2 triangles, otherwise it will need to be split again into two child nodes. On CPU, the algorithm to build the BVH is usually done recursively (which is the case here).

The method used to determine how to split a current node is based on the Surface Area Heuristic, which consists of minimizing the cost of splitting. I used a simple heuristic to determine whether a split was worth or not: $C_{SAH} = A_l * N_l + A_r * N_r$ where N refers to the number of primitives in the child node (l for left, r for right) and A corresponds to the area of the set of primitives contained in the child node.

Initially, the SAH cost was computed for every centroid of each triangle (primitive) in each direction (x, y, z), but as it was very costly I chose to subdivide uniformly the bounding box of the current node in each direction and test each slice instead. This allowed me to modulate the performance of the BVH construction which can be a lot since everything is executed sequentially on GPU using a recursive algorithm.

There are many ways to improve this step of the process (using binned BVH for example) but I did not have time to dive into it.

Collision detection on GPU

The detection of collision is done inside a kernel called ***COLLISION_DETECTION***. This kernel is called for every collider in the scene and for every direction of raytracing.

The test for collisions inside the kernel basically corresponds to that of a raytracing algorithm, using a top-down tree traversal for each ray (ie, for each vertex in the mesh as **1 ray = 1 vertex**). Instead of using a simple recursive algorithm I chose to go with an iterative traversal using a stack as recursivity can cause significant execution divergence between the different threads in the kernel. As written in the last section, the intersection test with each collider's BVH is executed on each vertex of the cloth's mesh, therefore the Grid dimensions and Block dimensions are respectively: $((N_{cloth})/32, (N_{cloth} + 31)/32, 1)$ and $(32, 32, 1)$. Each call to this kernel is followed by a call to CUDA_DEVICE_SYNCHRONIZE.

Inside the kernel, there are two main phases that need to be sorted out:
→ A broad phase, which consists in traversing the tree from the root towards the leaves and executing basic intersection tests between the current ray and the AABBs. Only internal nodes that pass the Ray-AABB test have their children nodes pushed onto the stack so that. When the loop come accross a leaf node, than the algorithm moves on to another phase: the narrow phase.

→ The narrow phase performs a Segment-Triangle intersection test on every leaf node reached in the broad phase. Using a Segment-Triangle intersection test allows to introduce a cloth thickness which mimics real life interactions. In practice, this test is much more costly than the simple AABB-Ray intersection and therefore needs to be executed only when necessary, that is why the BVH construction needs not to be too much simplified, otherwise the resulting tree will be poor and not optimized while performing intersection tests. Therefore, there is a clear trade-off between the time complexity spent at the beginning when building the BVH, and the one that is spent to traverse the BVH during the simulation. For this reason, building the BVH on the GPU would be a good way to have a quality tree in less complexity but this task is challenging and need more background work for it to work well.

→ For each leaf node that passes the narrow phase, the ID of the triangle and the ID of the collider concerned by the test are stored inside a buffer of (int, int). The intersection point on the triangle is also stored in another buffer for the next step: the collision response. From all the colliders and ray directions, only the closest intersection point is retained.

Collision response

Collision response starts when all colliders and ray directions have been treated by the collision detection algorithm. This phase consists in modifying the predicted position and velocity of the vertices obtained at the end of the Integration step part so that the cloth mesh does not penetrate the colliders.

For this step, I used the *penalty* method, which consists in adding penalty forces to the system that are supposed to represent the interaction between the cloth's mesh and the intersected object(s).

As this part is much more related to the computational physics course I will not dive deeper into the details of the implementation but this step mainly involves adding a reaction force that will keep the vertex from falling inside the collider, as well as a friction force based on the Coulomb model. The last force's intensity can be managed by a friction coefficient that causes the cloth to slide more slowly on the surface of the collider when having a high value, as Coulomb friction model aims to mimic the rough contact between the irregularities of each object's surface.

The penalty forces are added for every vertex that is involved in a collision, this verification is done by checking the content of the collision detection buffers. Overall, this step is done through a call to the kernel ***COLLISION_RESPONSE*** which has the same Grid and Block dimensions as the ***COLLISION_DETECTION*** kernel. Each call to this kernel is followed by a call to CUDA_DEVICE_SYNCHRONIZE.

III Results

A Existing features

From all the features I wanted to implement, I almost got them all done. However, there are of course many choices that could be improved regarding the use of the GPU as none of them used threads synchronization nor take advantage of the shared memory.

As a matter of fact, I did not find papers that managed to solved collisions on GPU using shared memory, this type of memory mainly benefited researchers that dealt with implicit numerical integration: the inversion of sparse matrices is a procedure that goes well with the use of shared memory whereas explicit integration is usually more suited for non-synchronized/independent kernels.

The only element for which I was planning to use the full GPU potential was for the optional feature: the cloth/cloth collisions. Unfortunately, I did not have time to implement properly this feature but I will try to describe the solution I started to build further in this section.

A - a Code profiling

It was overall quite hard to start a project only on GPU as I had no clue on how to debug correctly my program: one would argue that coding every element on CPU first would be a better idea as there are many user-friendly debuggers on VSCode or other IDE, but doing this would not have helped me understand how to debug my kernels properly. In addition, finding the CPU equivalent of the kernels I wrote is not a difficult task, but for kernels that involve more advanced GPU concepts this would be a waste of time. For this reason, I stayed on GPU even though I had to take some time to figure out how to use cuda-gdb properly.

Also, as this project is a personal project, I decided to work on my own GPU which is the NVIDIA RTX 3050. For information, I also tested my project on the NVIDIA RTX 3070 that ran the simulation around 15fps faster but the main device I worked on was the RTX 3050. This device as a global memory of 4GB and each block possesses at most 49kB of shared memory and around 65000 registers. The maximum threads per block is 1024, which is the number of threads that I used in all the kernels.

Also, since I decided to use CUDA to do the physics computations and not other API such as OpenCL, I also decided to use NVIDIA Nsight Compute which is a tool specifically designed to profile CUDA code on the system's GPU and see whether each kernel call was optimized or not. Indeed, making prints for each kernel can be quite a hassle and impractical when considering a large project.

In terms of performance, we use as a scene of reference a scene with a ground and a heart-shaped object which are 2 colliders respectively 4802 and 11264 triangles leads to BVH tree stored in arrays of 9604 and 22528 nodes. The mesh of the cloth is dropped on the heart-shaped object at the beginning of the scene. Based on this scene, Nsight Compute provides the following information:

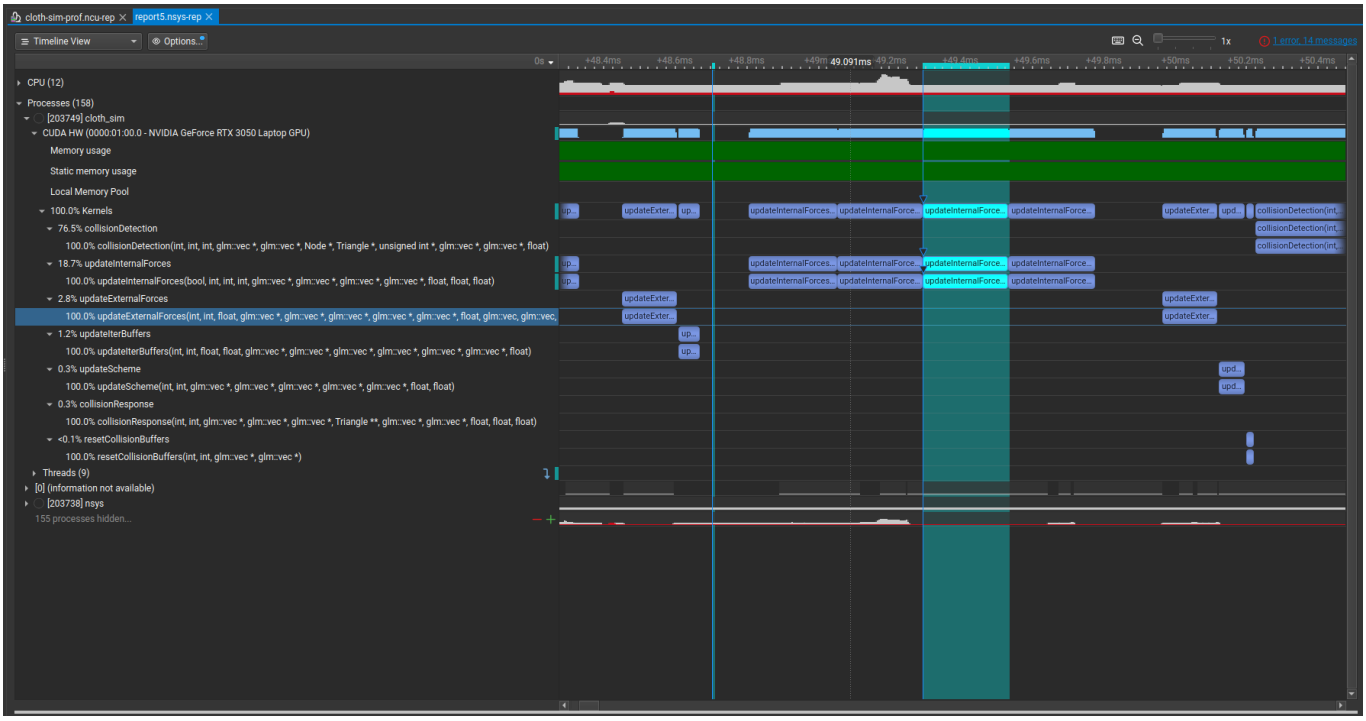


Figure 6: Capture of one of the panels displayed by the NVIDIA Nsight Compute tool (Timeline View)

A - b Performance analysis

With no surprises, the ***COLLISION_DETECTION*** kernel is the most costly: a single execution of the top-down tree sweep can go up to 20ms for the heart-shaped collider with more than 20000 nodes. This makes the framerate drop to 30fps, which is still real-time but not that great. In practice, the poor performance of this kernel is highly dependent on the memory and computation requirements of the tree traversal. For example, each call to this kernel allocate an array (the stack for the iterative traversal) which is by definition located in each thread's own local memory. Using such an array for writing/reading constantly on/from it is not ideal given that thread's local memory is much slower both in terms of bandwith and latency than either registers or shared memory. As a result, one might think about using the shared memory to store each thread's stack but this solution is not feasible at the moment given that each block has around a thousand threads and the shared memory capacity is only of (maximum) 49kB.

The performance of the stack-based traversal also depends on the quality of the BVH but heart-shaped object's BVH already took around 4s to be constructed on CPU even when simplifying the split decision process with a uniform subdivision, as explained in the BVH part before. Without optimizing the construction step of the BVH using binned-BVH or other methods, the tree quality cannot be augmented. If anything, improving the quality of the tree won't necessarily notably improve the performance of the overall kernel.

Comes after that the kernel responsible for updating the internal forces in the cloth which is probably due to the atomic operations. This performance issue could be solved by building a buffer for each type of spring, as explained in the corresponding section before.

B The missing feature: cloth/cloth collisions

The idea is to reuse the framework built for the cloth/object collisions to include cloth/cloth collisions, also called *self-collisions*.

It is important to note that until now, the algorithms used for cloth/object collisions suggested that the collider is a **static** object, meaning that it does not move in space. This point is particularly important as it implies that the BVH of the collider does not need to be updated each frame.

However, when dealing with self-collisions, we need to take into account the fact that the triangles of the collider (the cloth) will move over the frames. The first algorithm we can think about to update the BVH is to traverse the tree from root to leaves by updating the AABBS of the current internal node using an union of the children's updated AABB. However, this algorithm is not suited for our purpose as it cannot be parallelized easily for the GPU. Even if we made many threads update different part of the tree (subtrees) in the same time and combine the result, the execution divergence would still hamper greatly the performance.

One thing I noticed when searching for answers is that reduction patterns are often used to perform 'scan' (or depth-first search) operations on trees. The reduction pattern described in the GP-GPU course can actually be used to update balanced binary trees as it takes 2 elements at each step and combines them into a new elements. The only problem with that is that the BVH is not necessarily a balanced tree, meaning that some leaves are a level higher than the others in the tree.

I solved this problem by adding padding to make all leaves on the same level, as shown by the following figure:

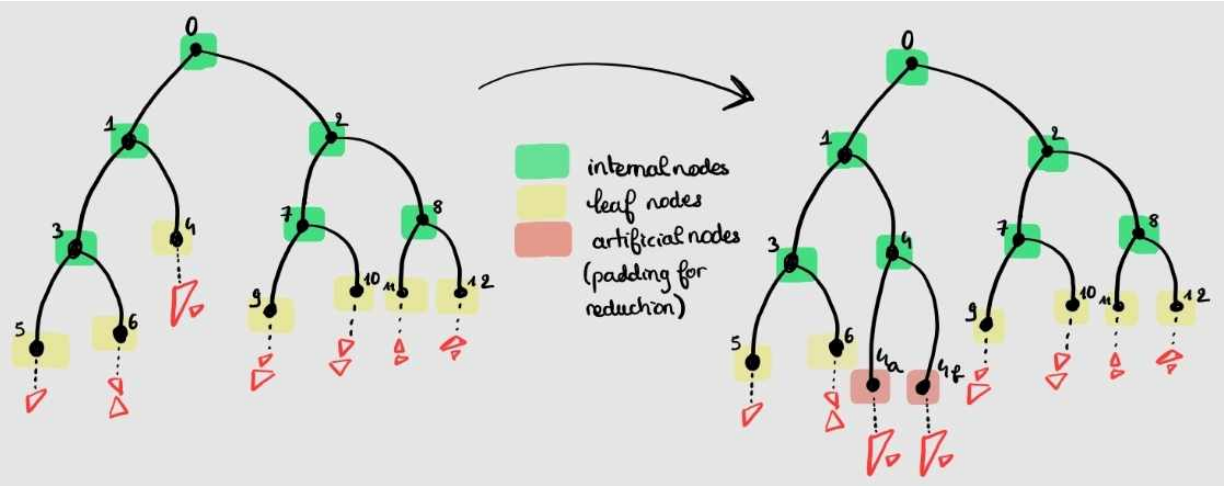


Figure 7: BVH Tree before and after the padding step. The new artificial leaf nodes are just copies of the original leaf node they come from.

After the padding step, we define an array for all the leaf (artificial or not) nodes' AABB. We know that the size of the array is always an even number as it corresponds to the number of leaves of the tree after padding. We can construct the array by separating the child from the same parent. Let n be the number of leaves in the tree (after padding): each right child AABB will be stored n/2 elements farther from its left sibling's. By storing the tree this way, we can use a more advanced reduction algorithm (described in details in the GP-GPU course) that allows coalesced access between threads, as shown in the below figure:

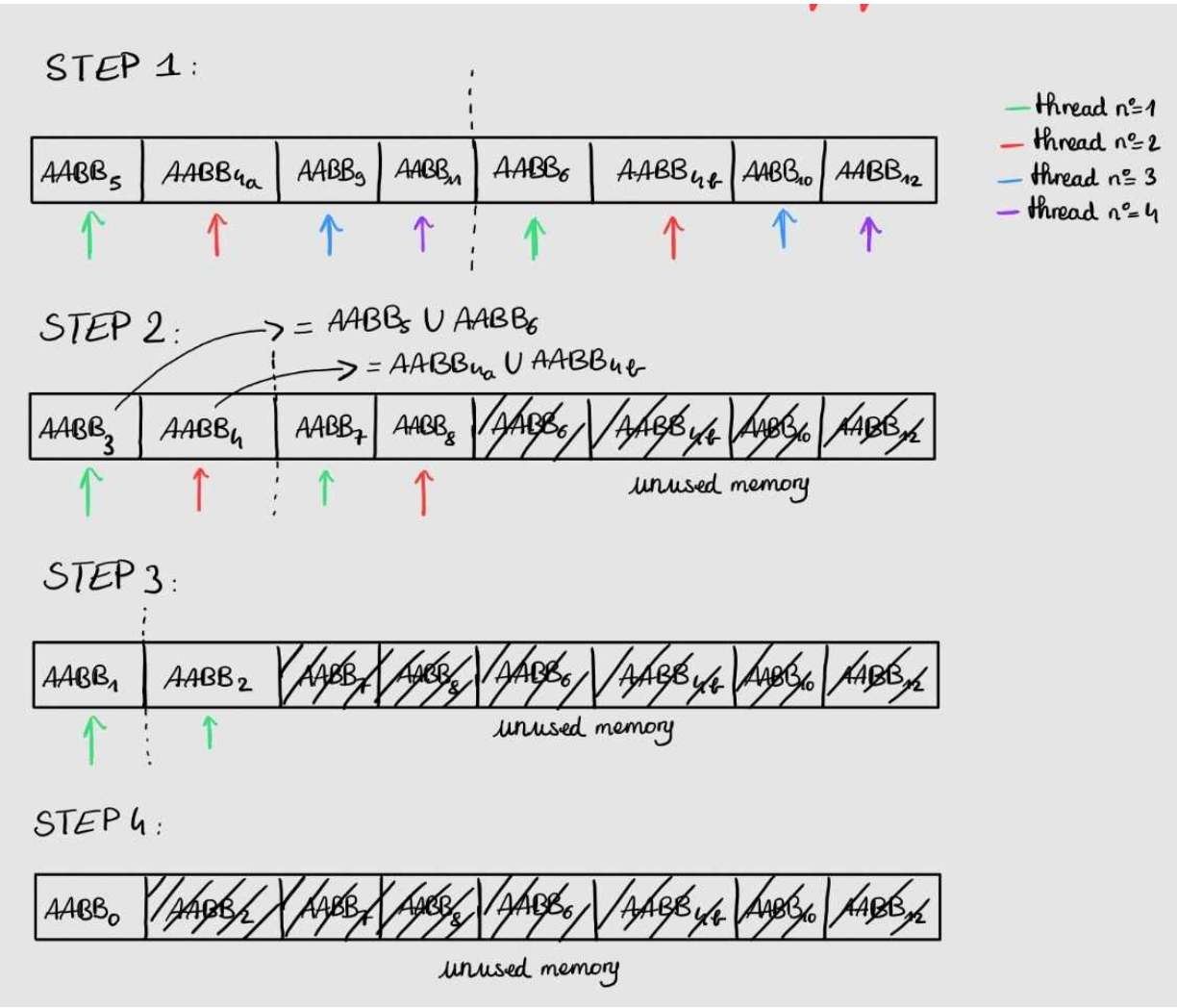


Figure 8: Scheme illustrating the reduction algorithm performed to update the BVH tree in the last figure. The AABB are numbered according to the node to which they are bound.

IV Conclusion

As a conclusion, this project was really fun to work on and was quite instructive. Even if I have not had time to finish entirely all the features I wanted to implement, I am still satisfied with the actual result. I do think that this choice of subject allowed me to use the actual content of the computational physics course as I had to apply direct knowledge of mechanics into my project, combined with advanced notions on GPU management thanks to the GP-GPU course.

I also had the opportunity to dive deeper into the field of computer graphics by approaching the subject of numerical collisions using BVH, which is something I wanted to approach for a long time.

Many improvements could be made for every step of the simulator, but I plan on pursuing it as it stands as a personal project for me.

Thank you for your reading!

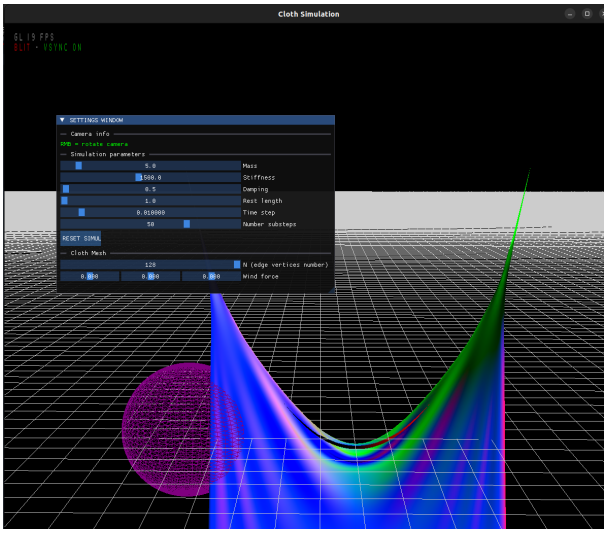
V Sources

Here are the main sources I used for this project, most of them being research papers on cloth simulation from researchers in computational physics.

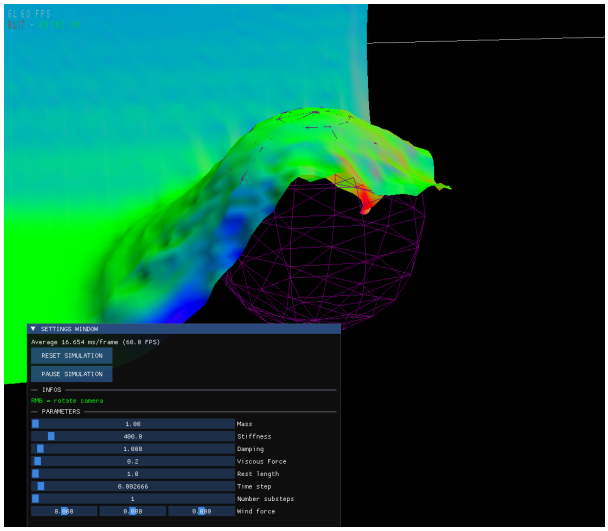
- Cloth model - https://escholarship.org/content/qt30g0h9r5/qt30g0h9r5_noSplash_3b2fd0df73a2pdf
- Cloth model - https://sci-hub.3800808.com/10.1145/280814.280821
- Cloth model - https://www.wuwayne.com/files/clothsim/report.pdf
- Collisions handling - https://d-nb.info/975583158/34
- Collisions handling - http://gamma.cs.unc.edu/gcloth/cloth.pdf
- n-body simulation - https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulationchapter-31-fast-n-body-simulation-cuda
- Oriented Bounding Boxes collisions - https://www.cs.upc.edu/~npelechano/paperCollDet02.pdf
- Collisions handling - http://www.r-5.org/files/books/computers/algo-list/realtime-3d/Christer_Ericson-Real-Time_Collision_Detection-EN.pdf
- Cloth model + Collisions handling - https://physbam.stanford.edu/~fedkiw/papers/stanford2002-0pdf
- BVH refitting - https://graphics.stanford.edu/~boulos/papers/togbvh.pdf
- BVH handling - http://gamma.cs.unc.edu/CSTREAMS/i3d.pdf
- BVH refitting - https://collections.lib.utah.edu/dl_files/4d/8c/4d8cce348efdadbf12e7ac0367pdf
- Continuous collision detection - https://sci-hub.3800808.com/10.1145/1342250.1342260
- Continuous collision detection + Coulomb friction https://graphics.stanford.edu/courses/cs468-02-winter/Papers/Collisions_vetements.pdf
- Parallel Prefix Sum - https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda?fbclid=IwAR2cDjvHNvsHVvT0c8znBULKSvKYPZUnIaWZ4-Lx8yklXGgPeEOOEI
- BVH handling/refitting - https://www.sci.utah.edu/~thiago/papers/AsyncBVHJournal.pdf?fbclid=IwAR1dnTTKHcPUMSdvOsQ1yxYmryxv9Mev4Y7GzwFPrHjjP06XuLbYCt1ymRk

• BVH traversal - https://l.messenger.com/l.php?u=https%3A%2F%2Fdeveloper.nvidia.com%2Fblog%2Fthinking-parallel-part-ii-tree-traversal-gpu%2F&h=AT2B-O_LaH-51i5d34AoLDqE7k-gr2HW0PKEOJcCaRJJ5l9yNPOTo2rjTWQXPP4NqvHZWCNzpG7-B0RXTECR87Sbm-DSw
• BVH introduction - https://jacco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics/

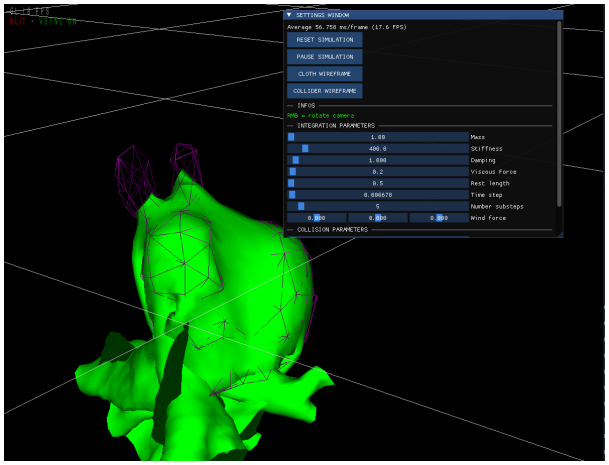
VI Gallery



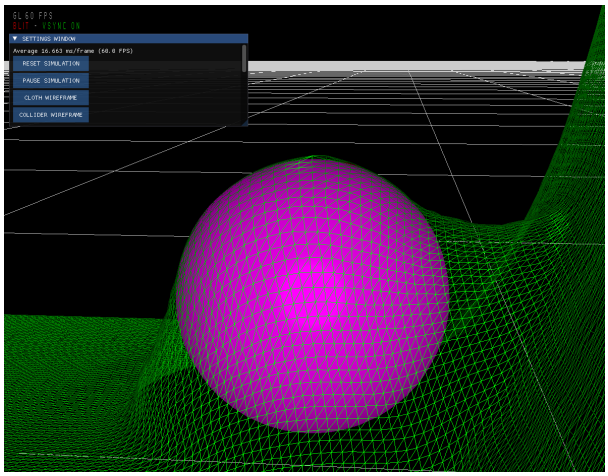
(a) Simulator with no collisions



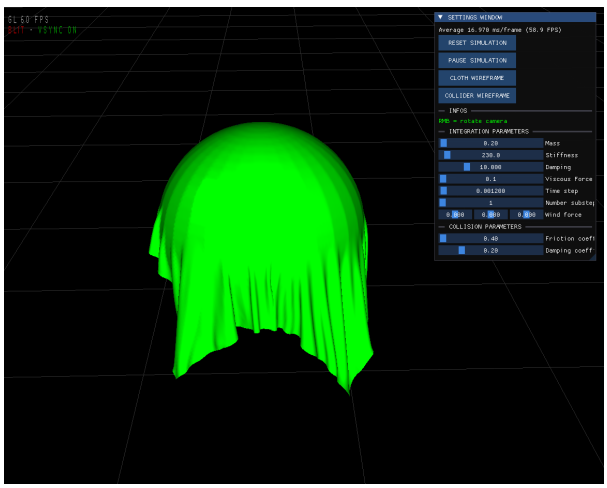
(b) Integration of collisions - beginning



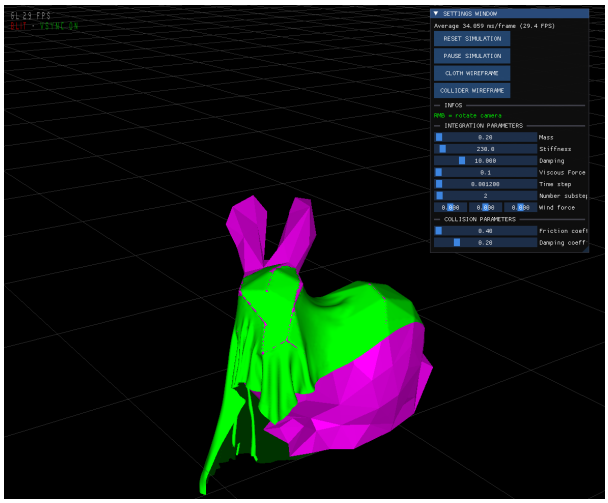
(c) Integration of collisions - bugs in the collision response



(d) Integration of collisions - debug of collision response using simple colliders

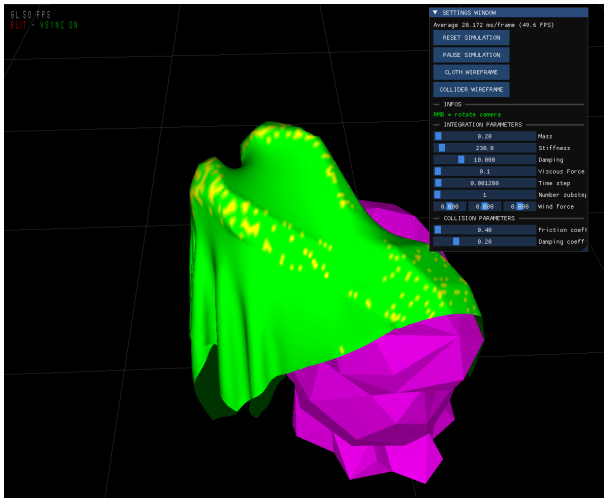


(e) Integration of collisions - some improvements but collider edges are harder to handle

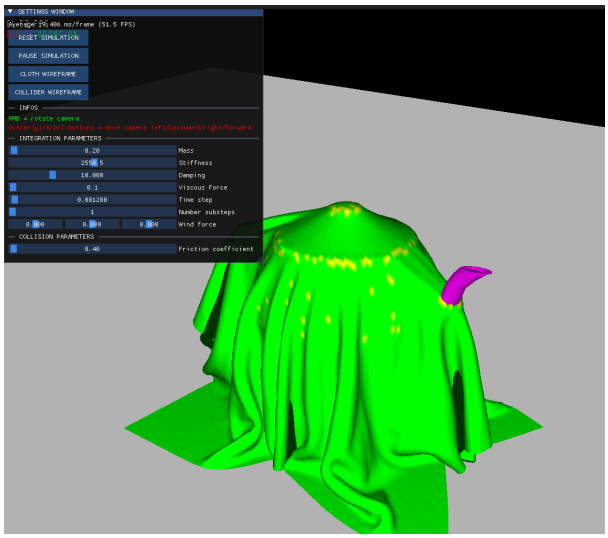


(f) Integration of collisions - debug using harder colliders (rough angles in the Stanford bunny)

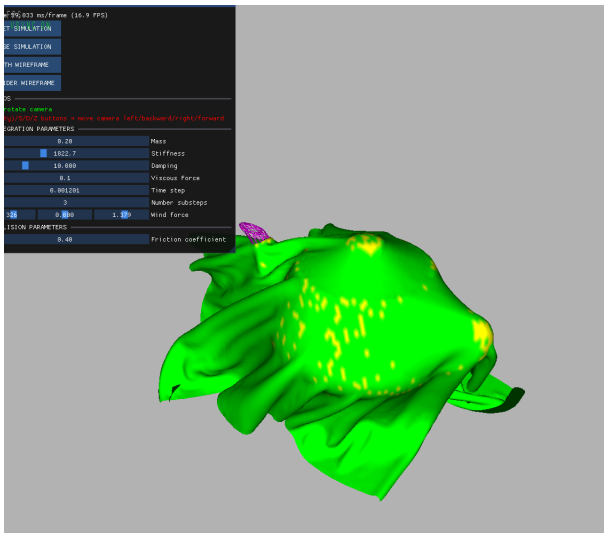
Table 1: Screenshots of the simulator (in chronological order)



(a) Stanford bunny - Yellow stains represent vertices that are indeed detected for collisions at the given frame

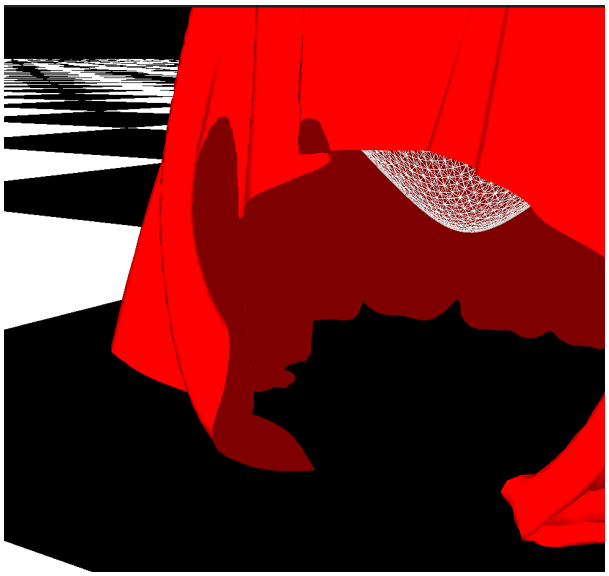


(b) Teapot - The opening of the teapot's neck is too thin to pass the collision detection test, unfortunately

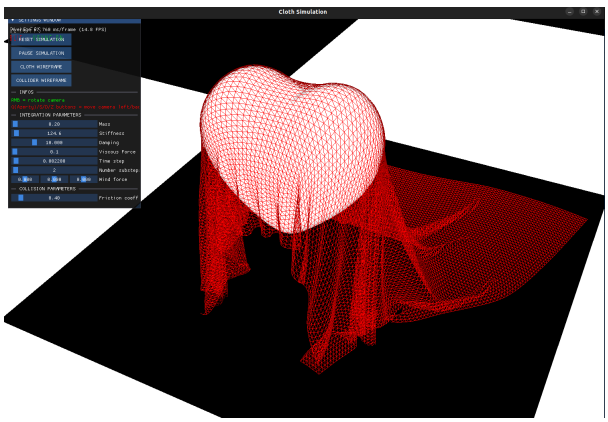


(c) Teapot with wind

(d) Scene of reference



(e) Scene of reference



(f) Scene of reference - Wireframe mode

Table 2: Screenshots of the simulator after correcting most of the bugs

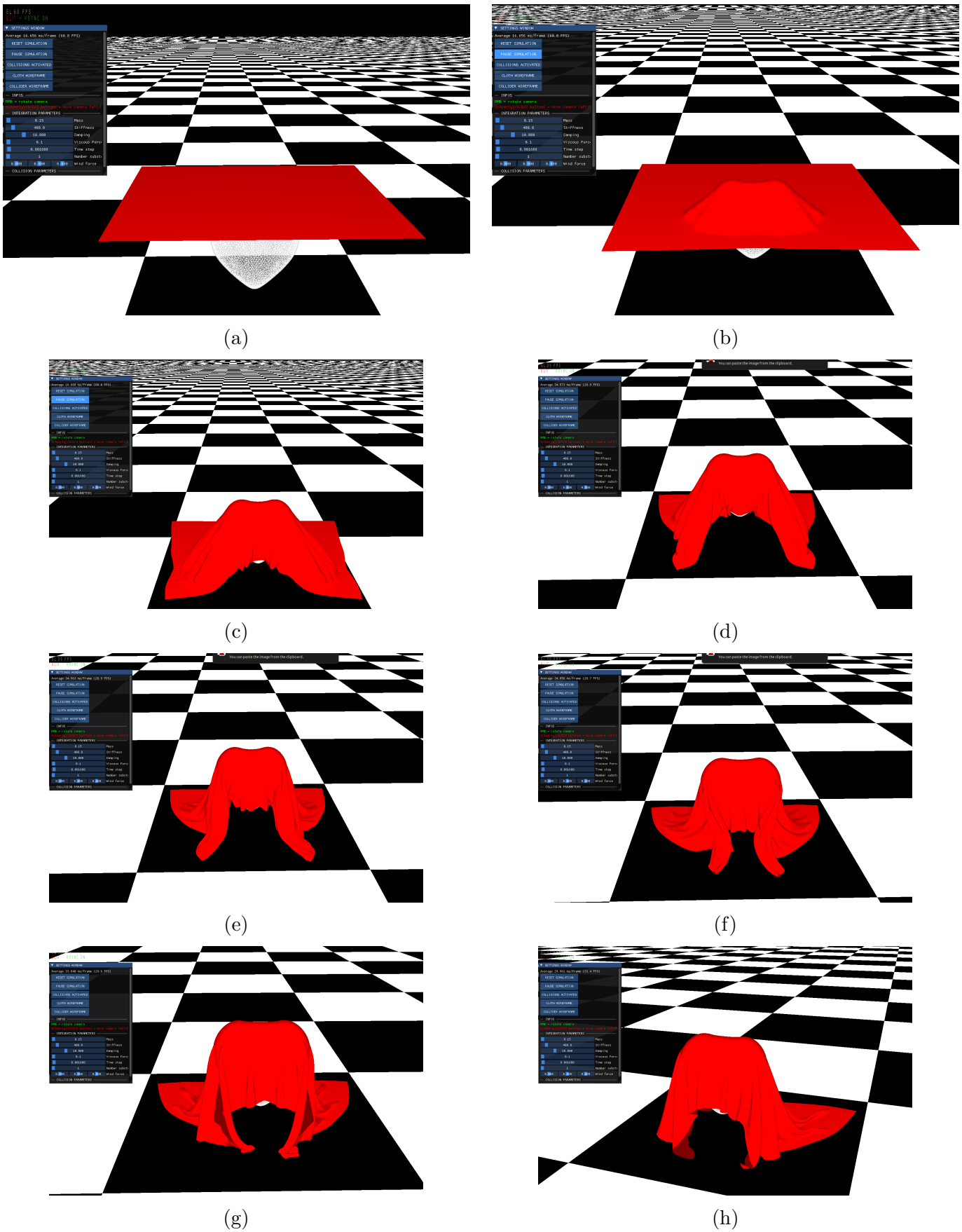


Table 3: Frame captures of the scene of reference in chronological order. (ground = 9604 triangles / heart = 11264 triangles / cloth = 16384 vertices)