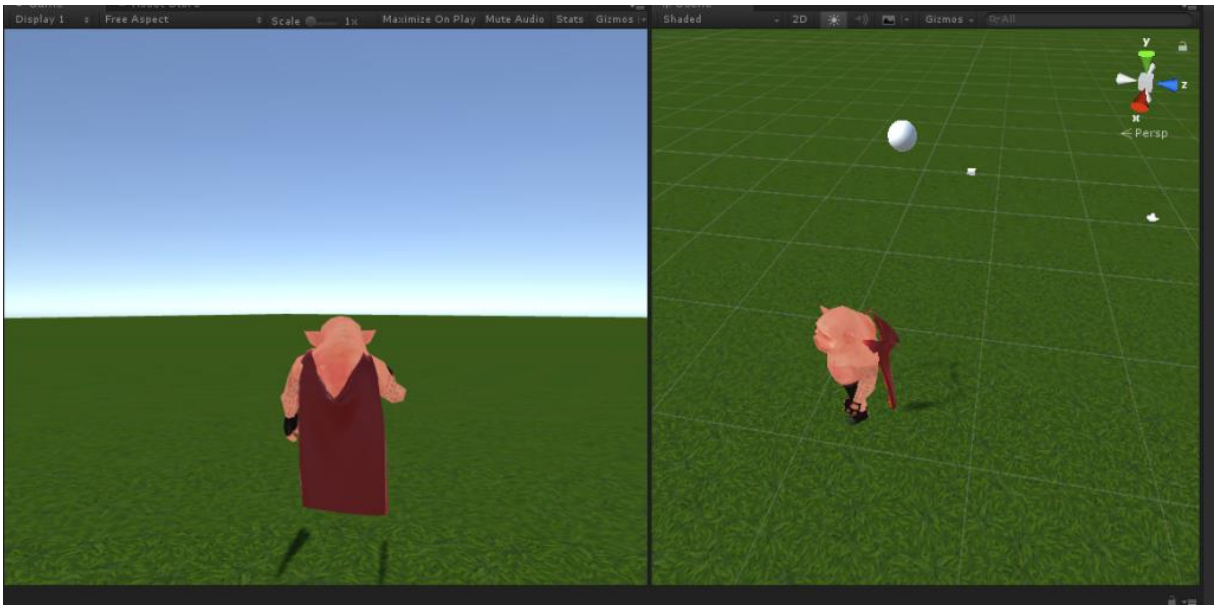
Cloth Cape Animation with Mass Spring Model

Oskar Norinder, Carl Svedhag

KTH Royal Institute of Technology, Brinellvägen 8, Stockholm, Sweden

Submitted June 10, 2019



Introduction

In this report, a step by step account for the development and implementation of an animated cloth cape will be presented. Based on the tasks of implementation specified in the project specification, an interactive program has been created using Unity Game Engine. However, to the extent found necessary for the correct completion of this project and course, prebuilt tools and packages of Unity was disregarded for. Such tools would certainly provide niftier solutions to the many challenges that were faced during the project work. Nonetheless, since the intent of this project was to learn about the methods and models by practical implementation, disregarding most prebuilt tools was sensible. The programming language used was C#.

Tasks established in the project specification, in order of highest priority, were: **(1)** implement a *Mass Spring Model* for the cloth cape animation, **(2)** implement a collision detection algorithm hindering the cape from intersecting with a moving game character, **(3)** apply wind forces to the cloth

cape, **(4)** implement a cloth collision detecting algorithm hindering the cape from self-intersecting.

Mass Spring Model

Theory

The Mass Spring Model is one of the simpler methods for animating cloth [1]. By approximation, a resting sheet of cloth can be described as a flat plane. Thus, when a Mass Spring Model is initialized, it is represented as a hypothetical 2D grid. Vertices of the grid are evenly spaced and interconnected by grid lines running parallel along the x- and y-axes. Furthermore, vertices are also diagonally connected to their neighboring vertices, resulting in a *King's Graph*, see *image 1*.

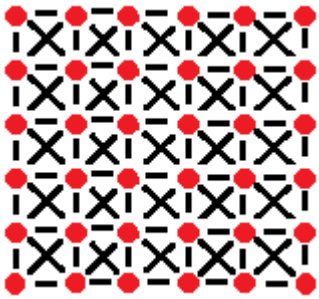


Image 1. The King's Graph with nm vertices and $4nm - 3(n + m) + 2$ grid lines.

In this implementation of the Mass Spring Model, a $20x20$ grid system is used. In theory, each grid vertex is considered to be a single particle within the sheet of cloth. Computer-simulated forces acting upon a particle will affect its trajectory and resulting position. Moreover, as the grid lines connecting each vertex to its neighboring vertices represent stretchable springs, forces acting upon one vertex will spread and affect its neighboring vertices. Spring forces distributed to neighboring vertices are computed via *Hooke's Law*.

Code Implementation

Two C# classes, representing a Vertex and an Edge (grid line) respectively, was firstly constructed. Upon executing the main script of the program, $20x20$ Vertex class objects are generated and saved in the array *Vertices* of size $20x20$. Each vertex in this array is initiated with a starting position in the grid, a constant mass property and a constant gravity force vector $\vec{G} = (0.0, -9.8, 0.0)$. The initialized position of a vertex in the 2D grid is calculated using its index in the Vertices array.

Furthermore, once all Vertex objects have been created and appended to the Vertices array, $4x10x20 - 3(10 + 20) + 2$ Edge class objects are generated and added to the C# list *Edges*. In an instance of the Edge class, the initialized vertices of which the Edge is connecting are saved. Additionally, using the initialized positions of these two vertices in the 2D grid, the length of the spring in its equilibrium state is computed.

Upon having created and initialized these two classes with their inherent conditions, Hooke's Law $F = k(spring_{rest} - spring_{stretched})$, with k being the spring constant, along with *Newton's Second Law of Motion* can easily be used to calculate the sum of forces acting upon one Vertex. Dividing this sum of forces with the mass of a single Vertex will consequently produce the acceleration vector $a = F \div m$, with the constant gravity force vector $\vec{G}$ firstly having been added to F. Lastly, after implementing a time step of 0.00875 seconds, approximately $1 \div 4$ of the renderers sampling rate, the acceleration vector is used to calculate the position of a Vertex in the upcoming frame.

To exemplify, we look at the Vertex in the Vertices array indexed at the first row, second column (0, 1). The position of its neighboring Vertex indexed at the first row, first column (0, 0) is unaffected by applied forces due to it being attached to the shoulder of the virtual game character, simulating a countering normal force. Once the main script is updated for the first time, 0.035 seconds after its initial execution, the new extension of the Edge connecting vertices (0, 1) to (0, 0) is used to calculate the resulting spring forces between the two via Hooke's Law. Since the previous frame, the extension of this Edge has only been affected by the constant force of gravity $\vec{G}$ being applied to the Vertex (0, 1), causing it to be pulled in the negative direction of the global y-axis.

Now, the sum of the spring force and the force of gravity being applied to Vertex (0, 1) is used in Newton's Second Law of Motion to calculate the acceleration vector of the Vertex. Using the specified time step and the acceleration vector, the new position of Vertex (0, 1), for the secondly rendered frame, is computed and set. Once a few iterations of this algorithm have been completed, Vertex (0, 1) should by approximation be directly underneath Vertex (0, 0). By this time, the increasing extension of the Edge connecting the two vertices has grown, resulting in a spring force great enough to cancel out the gravity force $\vec{G}$ being applied to Vertex (0, 1). The cape is hanging lose and Vertex (0, 1) is now static.

However, in this implementation Mass Spring Model, due to Vertex (0,1) having been affected by spring forces of other neighboring vertices, it's actual position after some iterations of the algorithm is not directly underneath Vertex (0, 0), see *Image 2*. Invisible springs connecting Vertex (0, 1) to vertices (0, 2), (1, 0), (1, 1), (1, 2) can be imagined by revisiting *Image 1*.

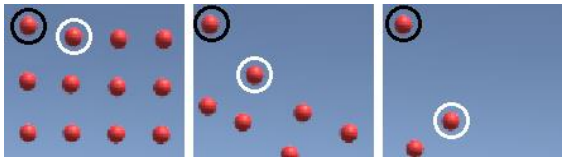


Image 2. From left to right, the amount of time steps has increased. Red spheres = vertices, circled black = Vertex (0, 0), circled white = Vertex (0, 1).

Collision Detection

When implementing a collision detection algorithm for the animated cape, several solutions were trialed. The idea was for the cloth cape to be realistically hindered by obstacles in the Unity scene. Furthermore, initial requirements were for the cloth cape to adapt in accordance with the obstacles size, shape and movement upon collision.

The trialed cloth cape collision algorithms of this project were varied, stretching from rather simple implementations to more advanced ones. However, due to the fact that the more advanced implementations did not result in any satisfactory visualization, a simpler collision detection algorithm was chosen for the final project deliverable. Moreover, the time constraint of this project was a determining factor for settling with a more simple solution. For the future development of this cloth cape animation, implementing a more advanced collision detection algorithm would be in the primary focus. Still, respecting the time spent on developing the more advanced, yet faulty collision detection algorithms of this project, these trialed solutions will also be briefly reviewed.

Code Implementation

For the implemented collision detection algorithm of this project, spheres are used for

detecting a cape collision. Upon executing the main script of the program, sphere GameObjects in the Unity scene tagged as *sphereCollide* are found using the *FindGameObjectsWithTag()* method of Unity. Every GameObject in the scene found by this method are saved in a new GameObject *collideSpheres*. The collideSpheres GameObject is then passed as an input variable to the *update()* method of every Vertex. As mentioned previously, all Vertex class instances are updated every 0.035 seconds.

To reiterate in simple terms, the *update()* method of a Vertex is used for calculating the new position of the Vertex. This is accomplished by summarizing the forces acting upon the Vertex class instance and converting this sum of forces into an acceleration vector. Lastly, the time step 0.00875 is used with the acceleration vector to determine the new position of the Vertex. The new position is finally set, causing the cloth to animate.

However, after having introduced spheres in the Unity scene with which a Vertex class instance should be able to collide, new constraints to the *update()* method were implemented for detecting collisions. Thus, the logic described in the previous paragraph is only followed up until the point of the new position of a Vertex being set. With colliding spheres in the Unity scene, this newly calculated position can only be regarded as a potential position. The 3D potential position of a Vertex class instance is therefore stored in the variable *potentialPos*. Before deciding whether the potential position should be final, a check need firstly be made to determine whether the Vertex has collided with a colliding sphere tagged as sphereCollide. By iterating through all colliding spheres of the collideSpheres GameObject, an if-statement is firstly used to determine whether the potential position of a Vertex is inside any colliding sphere. This is accomplished by measuring the distance between the potentialPos variable and every colliding sphere's center point in 3D. A comparison is then made between each measured value and the particular sphere's radius. If the radius' magnitude exceeds the measured distance, a collision has occurred.

After a collision of this nature has been recognized by the program, the potentialPos variable is overwritten and set to be the closest point on the colliding sphere's surface area, measured from the potential position. Lastly, after having exited the loop, the new position of the Vertex class instance is set to be the potentialPos variable. Thus, if the check is passed with no collisions detected, the position of a Vertex object will be updated normally.

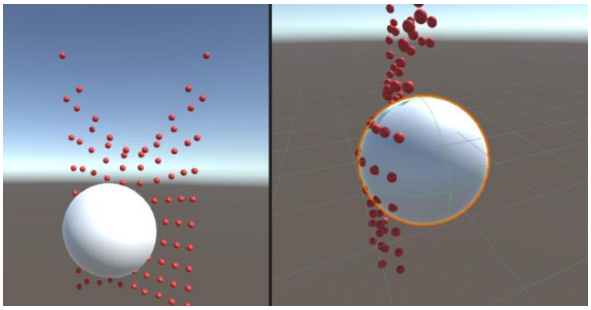


Image 3. Vertex objects of the cape colliding with a colliding sphere.

Alternative Solutions

Even though the aforementioned collision detection algorithm provided decent results in the case of colliding spheres, an eagerness for the implementation of a more generalized collision detection algorithm spurred the development of several alternative solutions. Although none of these solutions provided neither realistic nor stable results, a brief review of the algorithms will here be

presented. In part, this is due to the fact that the majority of the man-hours spent on this project were afforded to the development of these alternative collision detection algorithms.

The want for a finer, more generalized collision detection algorithm lead the project work to the development of an algorithm detecting collisions with polyhedron triangles. Since every mesh within the Unity scene are constituted by triangles, such an algorithm seemed optimal in regard to generalizability, enabling collision detection with any geometry. Using the previous position and the potential position of a Vertex object, an if-statement would be used to check whether a vector between the two positions was intersecting with a plane spanned by two corners of any triangle in the Unity scene. Barycentric coordinates would then be used to check whether an actual intersection with the triangle had been made. However, upon trialing such a solution, issues of using Unity's methods for translating between the local and global vector spaces of our program were encountered. Difficulties in retrieving the triangles of the prefabricated game character mesh[1] were also noted. Furthermore, additional problems arose as the suggested algorithm was found to be in demand of far too much processing power, causing the program to crash.

Consequently, since working with known geometries was assessed to be a more viable alternative in terms of processing power, work continued with the development and implementation of an *Oriented Bounding Box* (OBB) algorithm. However, in a later stage, while attempting to add the self-made bounding boxes to the game character mesh[1], the collision detection with boxes was found to provide an undesirable visual result. In hindsight, using a self-made cubical game character in the style of a board game meeple, as suggested in the project specification feedback, could have mended this problem.

Mesh Rendering [2]

For the main script to be executed on program launch, the script was added to an empty Unity GameObject *Cape* in the Scene. To generate the cape mesh, an empty MeshFilter component was in turn added to the Cape GameObject. Using the updatable Vertex class objects in the Vertices array, their positions are saved in an array *vertPos* and passed as vertices for the mesh. 3 positions in vertPos are used to constitute the corners of each mesh triangle. In a counterclockwise manner, index references to positions in vertPos are saved into an array *halftriangles*, see *Image 4*.

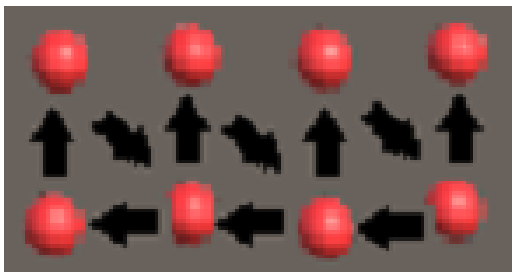


Image 4. Counterclockwise selection of mesh triangle positions

The array halftriangles is passed as triangles for the mesh. Lastly, the *RecalculateNormals()* method of the MeshFilter component is used to find the

[1] Lowpoly Pigman. Den Molina. Unity Asset Store. Retrieved from https://assetstore.unity.com/packages/3d/characters/lowpoly-pigman-75509 [Accessed 2019-07-01]

normal for each triangle in the mesh. The method is called after each position update of a Vertex class instance. Recalculating each normal is necessary for the proper shading of the mesh material, see *Image 5*.

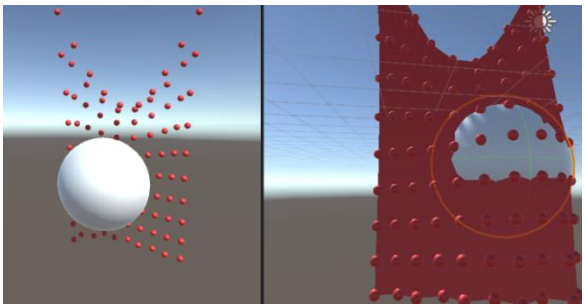


Image 4. Resulting one-sidedly rendered mesh.

At first attempt, this solution resulted in an one-sidedly rendered mesh due to the default property for flat mesh back-face culling of Unity being shut off. Consequently, attempts were made to double the number of triangles and iteratively construct them in a clockwise manner, opposite to what had been done with the halftriangles array. Even though this provided a solution to the problem of the mesh only being visible from one side, a prebuilt shader[2] for back-face rendering of flat meshes was chosen in favor of the unsatisfactory visual artefacts produced by this suggested solution.

Wind Simulation

Apart from the from cape animation being receptive to collisions and game character movements, an initial goal of the project had been to develop some sort of wind simulation to the cape animation. With the personal deadline of the project work drawing to a close, the simple solution chosen for simulating wind forces was a periodical sine wave. Thus, before updating the position of a Vertex class instance based on forces being applied to the Vertex and collisions, the entire Vertices array is passed as a parameter to the *update()* method of the class *Wind*. Within this method, a wind force is applied to each Vertex class instance. The wind force being applied to any particular Vertex is primarily based on the Vertex object's y index in the Vertices array. This, in turn, determines to what extent the sine wave should affect the position of the Vertex, at any particular update call. The direction and strength of a wind force being applied to a Vertex is somewhat randomized. However, the resulting wind simulation of the cape is perceivably wavy. Similarities to the animation can be found in a sheet of cloth hanging out to dry on a windy day. In the submitted project code, wind forces can be activated by uncommenting line 120 of script *Cape.cs*.

Wrap Up and Final Thoughts

After having developed the aforementioned algorithms, the cape was finally attached to two invisible colliding spheres positioned at the shoulders of the chosen game character mesh[1]. Adding several other colliding spheres to the bone structure of the prefabricated game character model, an early intention of animating the model's movements was additionally pursued. However, after trialing this, issues of understanding Unity's methods for translating between local and global vector spaces were yet again noted. Thus, in accordance recommendations provided in the project specification feedback, the

[2] UnityCoder. 2016. Unity Technolgies. Retrieved from https://gist.github.com/unitycoder/469118783af9d2fd0e2b36becc7dd347 [Accessed 2019-05-29]

decision of working with a static model was made.

Due to time constraints and a pressed schedule, several compromises to the program development were deemed necessary in order for us to finish this project in time. As previously mentioned, one such compromise regarded the collision detection algorithm of the project, which could have been enhanced in several alternative ways. Furthermore, due to this project being inspired by the flag project [3] of 2015, an initial expectation was to implement a similarly realistic wind simulation algorithm for this project. In the end, we had to settle with a less visually pleasing solution. A further exploration of the subject, with cleaner algorithms, could be found by implementing the methods and algorithms presented in the article by Lewin [4] from 2018. This research presents State of the Art cloth animation implementations in video games with many facets of cloth interaction. Lewin [4] does review cloth self-collision algorithms, the development of which had been a low-prioritized goal of this project. After having had a first glance at the research surrounding cloth self-collision, most algorithms did seem worryingly difficult to implement. However, in future work, after re-implementations and refinements to the existing solutions and algorithms of this project have been made, developing a cloth self-collision would make a good next step.

References

[1] Matthew Fisher. 2014. Matt's Webcorner: *Cloth*. Retrieved from https://graphics.stanford.edu/~mdfisher/cloth.html [Accessed 2019-06-03]

[2] Jasper Flick. Catlike Coding: *Procedural Grid*. Retrieved from https://catlikecoding.com/unity/tutorials/procedural-grid/ [Accessed 2019-07-01]

[3] Albin Lindskog. 2015. Spring Mass Model Flag. Retrieved from https://smmflag.wordpress.com/ [Accessed 2019-07-02]

[4] Chris Lewin. 2018. Cloth Self Collision with Predictive Contacts. Retrieved from https://media.contentapi.ea.com/content/dam/eacom/frostbite/files/gdc2018-chrislewin-clothselfcollisionwithpredictivecontacts.pdf [Accessed 2019-07-02]