BACHELORARBEIT

CLOTH SIMULATION FOR THE
VIENNA PHYSICS ENGINE

Verfasser

Felix Neumann

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2023

Studienkennzahl lt. Studienblatt:	A 033 521
Fachrichtung:	Informatik - Medieninformatik
Betreuerin / Betreuer:	Dr. Helmut Hlavacs

Abstract

This work describes and analyses a software project. Furthermore, an explanation of usage is presented. The software project is a real-time cloth simulation. It is not a stand-alone application but an extension for the Vienna Physics Engine. The Vienna Physics Engine is a simple, educational software which was created to support courses on videogames at the Faculty of Computer Science of the University of Vienna. The design of the cloth simulation is based on a literature analysis presented in this work. The software does not introduce a new or improved method for simulating cloth but implements a well-established method called Extended Position Based Dynamics. In addition, it extends the render engine of the Vienna Physics Engine, so that it can render simulated cloth. The implementation was written in C++. Most of the logic of the cloth simulation is contained by three new classes while the expansion of the render engine is mostly achieved by extending and adding logic to existing classes. The three main classes of the cloth simulation represent mass-points, constraints, and cloth. Known issues are being presented as well as potential solutions if possible. Furthermore, possible improvements are discussed.

Table of Contents

List of Code Blocks

List of Images

List of Tables

1 Introduction

1.1 Problem Statement

The Vienna Physics Engine [1] is a simple physics engine for educational purposes. It is used to support some courses on videogames at the Faculty of Computer Science of the University of Vienna. It allows students to simulate the behavior of rigid bodies and to create simple videogames. Rigid bodies are bodies whose shape cannot be transformed in a non-affine way. This means, that they can get translated, skewed, scaled, and rotated but for example not squeezed like a balloon. Bodies that can be transformed in non-affine ways are named soft bodies. Cloth, pieces of metal or inflatable things are commonly implemented to behave like soft bodies in more sophisticated physical simulations.

The Vienna Physics Engine can be seen as an extension for the Vienna Vulkan Engine [2], a render engine used by the Students of the Faculty of Computer Science of the University of Vienna to learn Vulkan. Vulkan is a multi-platform API for three-dimensional computer graphics. The Physics Engine makes use of the Vienna Vulkan Engine to render the simulated objects on screen. The Vienna Vulkan Engine does not support the rendering of objects which shapes change in non-affine ways and therefore, also does not allow for rendering soft bodies.

The software project of this work aims to extend the features of both the Vienna Physics Engine and the Vienna Vulkan Engine so that they support cloth simulations, a type of soft body simulation.

1.2 Goals

The first goal of this work is to design and implement an extension for both the Vienna Physics Engine and the Vienna Vulkan Engine that allows its users to create simple simulated pieces of cloth. The goal is not to invent a new or improved way of simulating and rendering cloth but rather to implement existing concepts within the context of the engine.

The second goal is to present an explanation of usage and to analyze and discuss issues and possibilities for future work. This should enable students to use the solution for their games as well as to improve and extend the simulation itself.

1.3 Methods

The design of the cloth simulation system is based on the findings of a literature analysis. The implementation is influenced by the preexisting C++ code of the Vienna Physics and Vienna Vulkan Engine and is written in C++ as well. The development time of the software project amounts to about fourteen weeks with an estimated average weekly time effort of ten to fifteen hours. The software was analysed by using smoke and performance tests. Problems concerning the implementation that have been found are presented by this work along with possible solutions. To check whether the goals were met, cloth simulated by the software was visually compared with real cloth and performance testing was done to confirm that the real-time requirement is met.

1.4 Preamble

To achieve stability and avoid bugs seasoned methods are needed for both the simulation and rendering aspect of the software project.

Data objects that represent a three-dimensional model and that are supposed to be drawn using Vulkan typically have an index and vertex buffer. Buffers are arrays of data that can be bound to a graphics card so that it can use the data for its calculations [3]. A vertex buffer stores data concerning the vertices of a models such as their positions, colours and normals. An index buffer consists of indices which act like pointers to the entries of the vertex buffer and allows for restructuring its entries [4].

1.5 Literature Concerning Cloth Rendering

For rigid bodies the vertex and index buffers are typically created when their models are loaded and not modified afterwards. Affine transformations are achieved by multiplying the position data of every vertex with a 4x4 matrix defining the desired transformation. Soft bodies, however, need a dynamic vertex buffer since their models are transformed in non-affine ways. There is very little literature on how this can be achieved using Vulkan. Diamond proposes to use a vertex staging buffer in addition to the vertex buffer [5]. Every time vertex positions change, the updated data are copied into the staging buffer via a memory mapping and then copied from there into the vertex buffer. The Vulkan documentation indicates that when creating the staging buffer, the `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` flag should be set so that the buffer can be used as the source of a memory transfer command [6]. According to Vulkan Tutorial [7], a popular website for learning Vulkan, the property flags `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` should be set when creating a staging buffer. This ensures that the memory can be mapped and does not need to be flushed manually [8]. The Vienna Vulkan Engine uses the Vulkan Memory Allocator [9], a memory allocation library. When creating a buffer by using this library the result of setting the `VMA_MEMORY_USAGE_CPU_ONLY` flag is equivalent [10].

Cloth models in computer graphics are typically infinitely thin surfaces [11]. In the real-world cloth is made from yarn which gives it highly textured and complex surfaces. While there are ways to replicate this look by generating high-polygon meshes with fibre-level details [12] [13] [14], this leads to very high computational costs even for offline applications [11]. To achieve a realistic look without using highly detailed models, view dependent effects such as occlusion and self-shadows and illumination effects like specularity and interreflection need to be simulated [15]. Coarse fibre level details can be created on the fly in real-time by using tessellation-shaders and yarn-level control points [11]. This method allows to further improve the level of realism by colouring the fibres with precomputed textures that simulate a bundle of thinner fibres and shading with precomputed self-shadows. The reflective properties of cloth can be simulated at interactive rates using a bidirectional texture function and generating view-dependent texture maps [16].

1.6 Literature Concerning Cloth Simulation

A lot of research exists on soft body simulation and cloth simulation specifically. Some methods are designed for offline others for real-time applications in mind [17, pp. 2-3]. Videogames and their engines are required to run at real-time. Early attempts at simulating cloth [18] used a geometric method suitable for one-frame renders but not for video-based media and behaviour simulation. Around the same time, the idea of using mechanical concepts like elasticity theory to simulate soft bodies was explored [17, p. 4]. A commonly used physically based method for simulating soft bodies is to use mass-spring models [19]. As the name suggest, this method represents models as a collection of mass points which are connected by virtual springs [17, p. 21]. The springs transfer forces onto the points which then leads to their velocities changing. The underlaying calculations are solved at discrete time steps [19]. A drawback of this method is that it tends to become unstable when an explicit integration scheme and stiff springs are used while implicit schemes typically require more computations and memory [20]. An alternative to the mass-spring model is to use Position Based Dynamics [21]. This method takes a position-based approach, meaning instead of applying forces to the mass points, their positions are changed directly, and the changes of their velocities are derived. The mass points in this system are usually referred to as particles. Instead of springs the method is based on constrained dynamics simulations [22] [21]. With Extended Position Based Dynamics [23] an extension of this method was introduced, which allows for implicitly simulating arbitrary elastic energy potentials.

A soft body can potentially collide with itself, rigid bodies, or other soft bodies. To check, whether a soft body penetrates a rigid body, ray casting can be used [24]. If the mesh of the rigid body consists of triangles, this can be done very efficiently [25]. When using a mass-spring model, collisions can be resolved by applying a collision reaction force [26]. When using Position Based Dynamics, collisions with rigid bodies can be resolved in a simple manner by projecting the particles to valid points if a penetration is detected [21]. This method also eliminates overshooting problems. Using Position Based Dynamics in combination with the method proposed by Baraff et al. [27] also allows for resolving cloth self-collisions [21]. To achieve a real-time performance with multiple soft and rigid bodies in a scene, using a broad and narrow collision detection phase [28] is advisable.

2 Design

2.1 Overview

The cloth simulation extension can be conceptually divided into two decoupled parts. The rendering and the simulation of cloth. The design of the simulation is mainly based on the findings of the literature analysis while the rendering aspect adapts and expands the system of the Vienna Vulkan Engine.

The physics and rendering parts are connected as follows. The engine class which manages all the functionalities of the software can create an instance of the class that contains all the code related to the physics simulation. A piece of cloth in the physics world has got an owner. The owner is expressed through a void pointer which should point to a cloth entity instance of the Vienna Vulkan Engine. This way, cloth instances can be accesses from within the Vienna Vulkan Engine by asking the class containing the physics simulation to return a pointer to the cloth instance corresponding to a specific owner. Furthermore, two callback functions can be passed to cloth instances of the physics world. One that gets triggered when cloth moves and one for when a piece of cloth gets deleted. These callback functions allow the Vienna Vulkan Engine to synchronize the cloth entities with the simulation. If a piece of cloth moves in the simulation, the callback function gets called and the Vienna Vulkan Engine updates the vertices of the corresponding cloth entity. If a piece of cloth gets deleted in the simulation, the cloth entity and potentially its child nodes get deleted as well.

2.2 Rendering of Cloth

The Vienna Vulkan Engine renders scenes which consist of hierarchically structured scene nodes which can be transformed in space. Cameras, lights, and entities are scene nodes. An entity possesses a pointer to a mesh and a material which when combined result in the model that gets rendered. Meshes and materials are managed by a scene manager. Two different entities in the scene can share the same mesh since for rigid bodies the positioning in space is defined by a model matrix and not the mesh itself. To allow for the rendering of cloth, a new cloth entity type was added. The mesh of a cloth entity is not managed by the scene manager but by the cloth itself. If two pieces of cloth load the same three-dimensional model, they both receive their own copy of the mesh. This way, the vertices of a cloth mesh can be modified without influencing the meshes of other pieces of cloth.

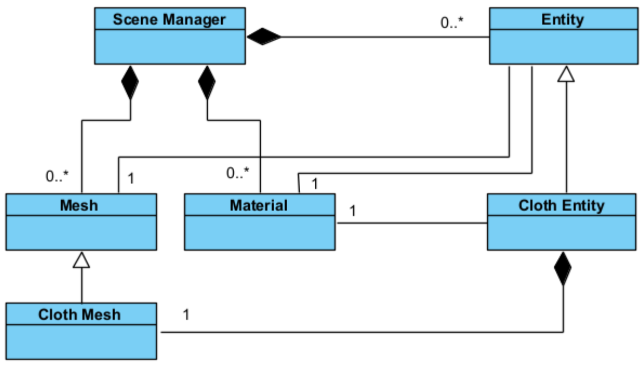


Image 1 Class Diagram showing how a cloth entity manages its own mesh.

Cloth entities require a special type of mesh. A cloth mesh extends the mesh class of the Vienna Vulkan Engine by the ability to update its vertices and a few cloth-specific requirements. In addition to the vertex and index buffers, it has got a staging buffer as proposed by Diamond [5] which was set up similar to the example of the Vulkan Tutorial [7]. The cloth mesh offers a function that can be used to update the list of vertices. This function firstly copies the updated vertices into the staging buffer and then from there into the vertex buffer.

The normal of every vertex is recalculated before it gets copied into the staging buffer. This is necessary since the orientations of the triangles of the mesh change if the positions of the vertices are modified.

Only flat-shading normals are calculated since smooth-shading requires more calculations. Additionally, a copy of every vertex is appended to the end of the list with its normal inverted. The list of indices, that gets copied into the index buffer when a cloth model is loaded, is also twice its original size. The second half is modified so that the triangles which the appended vertices form are drawn flipped. This prevents that only one side of a piece of cloth casts a shadow. This would otherwise happen since the Vienna Vulkan Engine only considers one side of a face for generating the shadow map. There is a dedicated cloth fragment shader program which ensures that the duplicated flipped triangles do not result in shadows on the cloth being drawn twice.

2.3 Cloth Simulation

The cloth simulation uses Extended Position Based Dynamics [23]. This part of the software is thus mainly not original work. The design is fully based on the work of Macklin et al. if not indicated otherwise. The simulation works as follows:

A piece of cloth in the physics simulation consists of mass-points and constraints. For each unique vertex-position of the underlying three-dimensional model there is one mass-point. A mass-point has got a position, previous position, velocity, and mass property. A constraint is a condition that the system must fulfil or try to fulfil. Each constraint involves two mass-point. Two types of constraints are used for the simulation: Edge and bending constraints. They are both distance constraints and only differ regarding the spatial relationship of their associated mass-points. A distance constraint has got a rest-length and compliance attribute and pointers to two mass-points. Compliance is the opposite of stiffness. To solve a constraint, the difference between the current distance of the two affiliated mass-points and the rest-length is calculated. The positions of the mass-points are then corrected so that the difference is decreased. The respective correction vector depends on the mass of a mass-point, the compliance of the constraint and the delta time of the step of the simulation. There is an edge constraint between the mass-points of every edge of the model and a bending constraint between the opposing mass-points of every pair of adjacent triangles.

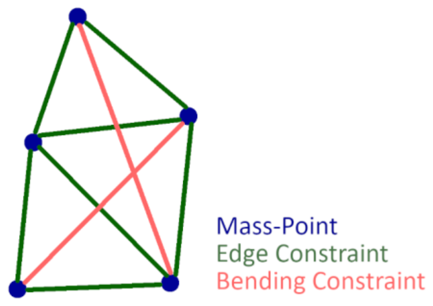


Image 2 Construction of edge and bending constraints.

Each step of the simulation the integrate function is called for every piece of cloth. The following section describes how its algorithm works conceptually.

```
update list of nearby rigid bodies.
for each substep
    for each mass-point
        apply external forces (damping, gravity)
        change position according to velocity
    for each constraint
        solve constraint
    for each mass-point
        resolve collisions with nearby rigid bodies
```

Code Block 1 Cloth integration.

The handling of collisions with rigid bodies is not based on the Extended Position Based Dynamics system proposed by Macklin et al. The Vienna Physics Engine uses a grid to do broad phase collision checking between rigid bodies. The cloth simulation makes use of this grid to create a list of nearby rigid bodies with which a piece of cloth could potentially collide. The narrow phase algorithm works for collisions between cloth and convex rigid bodies. It is performed for every mass-point.

```
for each rigid body nearby
    transform mass point position into rigid body local space
    for each face of rigid body
        find distance t from mass-point to face plane
        return if t < 0 (no collision)
        store projection point if it is the nearest so far
    store current mass-point position as previous position
    set mass-point position to projection point transformed into global space
    update velocity
```

Code Block 2 Cloth/rigid body collision handling.

Conceptually this algorithm checks whether the mass-point is behind all faces of a rigid body and changes its position to the nearest projection onto a face of the body if it is. Self-collision and collisions between cloths are ignored. Also, cloth treats the rigid bodies as static, and no impulses are applied to them.

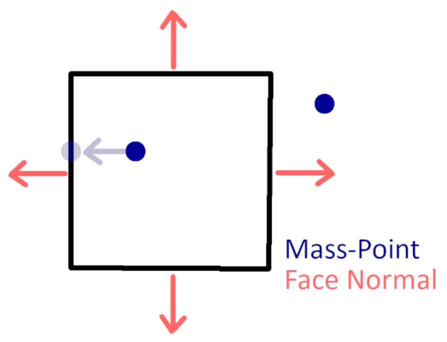


Image 3 If a mass-point is behind all faces, it gets moved.

Pieces of cloth can be moved by applying transformations. Every cloth can have a set of fixed mass-points which simulate points where the cloth is attached to some rigid object. These mass-points are not affected by collisions with rigid bodies. If a transformation is applied, it is applied to these fixed points and the other points are dragged along by the simulation. To prevent excessive stretching of cloth, a constant ranging from zero to one can be defined, which causes the unfixed points to be transformed as well but then sets their position to an interpolation between their original and new position so that they still get dragged along.

The creation of cloth (for explanation of usage see section 3.5) in the simulation requires a list of vertices and indices which represent the mesh. Firstly, mass-points are generated from vertices. For a set of vertices which share their position, only one mass-point is created. Every mass-point has got a list with the indices of all associated vertices. This is necessary so that an updated list of vertices which correlates with the original list can be generated. Secondly, the list of indices is used to organize the mass-points as triangles. This allows for creating the distance and edge constraints. Thirdly, the largest distance between two arbitrary mass-points is calculated and stored for collision checking. If a rigid object is further away from any mass-point than this distance - plus some margin that accounts for possible stretching - it cannot collide with any other mass-point. Lastly, a very small rotation is applied to the created piece of cloth to avoid that all mass-points lie on a plane which could rob the simulation of a dimension if this cloth is only translated exactly along the plane.

3 Implementation

3.1 Rendering of Cloth

3.1.1 Cloth Entity and Mesh

`VESceneManager` was extended by a function `loadClothModel` which allows for loading a model so that it can be used for simulating cloth. It creates an instance of `VEClothEntity`, `VEClothMesh` (which is managed by the former), and `VEMaterial` for every material of the loaded model. `VEClothEntity` extends `VEEntity` and conceptually only differs in its mesh type. `VEClothMesh` extends `VEMesh`. It creates the index buffer with the same function that is used for rigid bodies and the vertex, and vertex-staging buffer with a custom function according to the findings of the literature analysis (see section 1.5).

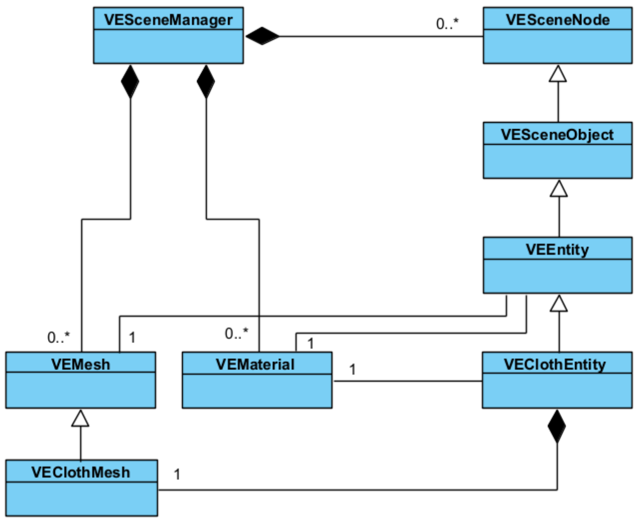


Image 4 Class Diagram showing new classes.

```
VkResult vhBufCreateClothVertexBuffers(VkDevice device,
       VmaAllocator allocator, VkQueue graphicsQueue, VkCommandPool commandPool,
       std::vector<vh::vhVertex>& vertices, VkBuffer* vertexBuffer,
       VmaAllocation* vertexBufferAllocation, VkBuffer* stagingBuffer,
       VmaAllocation* stagingBufferAllocation, void** ptrToStageBufMem,
       VkDeviceSize* bufferSize)
{
       *bufferSize = sizeof(vertices[0]) * vertices.size();

       VHCHECKRESULT(vhBufCreateBuffer(allocator, *bufferSize,
              VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VMA_MEMORY_USAGE_CPU_ONLY, stagingBuffer,
              stagingBufferAllocation));

       VHCHECKRESULT(vmaMapMemory(allocator, *stagingBufferAllocation, ptrToStageBufMem));

       memcpy(*ptrToStageBufMem, vertices.data(), (size_t)*bufferSize);

       VHCHECKRESULT(vhBufCreateBuffer(allocator, *bufferSize,
              VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT |
              VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
              VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
              VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT, VMA_MEMORY_USAGE_GPU_ONLY,
              vertexBuffer, vertexBufferAllocation));

       VHCHECKRESULT(vhBufCopyBuffer(device, graphicsQueue, commandPool, *stagingBuffer,
              *vertexBuffer, *bufferSize));

       return VK_SUCCESS;
}
```

Code Block 3 Function creating a cloth's vertex and vertex staging buffer.

The memory mapping to the staging buffer is only unmapped when the cloth mesh is destroyed.

As described in the Design section (see section 2.2), the lists of indices and vertices of cloth models need to be extended, before they are copied into the buffers, so that the back of the cloth correctly casts a shadow.

```
std::vector<vh::vhVertex> VEClothMesh::createVerticesWithBack(
       const std::vector<vh::vhVertex>& vertices)
{
       std::vector<vh::vhVertex> verticesWithBack(vertices);

       verticesWithBack.reserve(vertices.size() * 2);
       std::copy(vertices.begin(), vertices.end(), std::back_inserter(verticesWithBack));

       for (size_t i = vertices.size(); i < vertices.size() * 2; ++i)
              verticesWithBack[i].normal *= -2;

       return verticesWithBack;
}
```

Code Block 4 Function creating extended vertex list for correct shadow casting.

```
std::vector<uint32_t> VEClothMesh::createIndicesWithBack(std::vector<uint32_t>& indices,
       uint32_t highestIndex)
{
       std::vector<uint32_t> indicesWithBack(indices);

       for (size_t i = 1; i < indices.size(); i += 3)
       {
              indicesWithBack.push_back(highestIndex + indices[i + 1]);
              indicesWithBack.push_back(highestIndex + indices[i]);
              indicesWithBack.push_back(highestIndex + indices[i - 1]);
       }

       return indicesWithBack;
}
```

Code Block 5 Function creating extended index list for correct shadow casting.

Every time the vertices of a simulated cloth change, the data in its vertex buffer needs to be updated accordingly. The index buffer remains unchanged. The bounding sphere can optionally be updated which is done by finding the center of the mesh and then the longest distance from there to a vertex.

```
void VEClothMesh::updateVertices(std::vector<vh::vhVertex>& vertices,
       bool updateBoundingSphere)
{
       updateNormals(vertices);

       vertices = createVerticesWithBack(vertices);

       VECHECKRESULT(vh::updateClothStagingBuffer(vertices, m_bufferSize,
              m_ptrToStageBufMem));

       VECHECKRESULT(vh::updateClothVertexBuffer(
              getEnginePointer()->getRenderer()->getDevice(),
              getEnginePointer()->getRenderer()->getVmaAllocator(),
              getEnginePointer()->getRenderer()->getGraphicsQueue(),
              getEnginePointer()->getRenderer()->getCommandPool(),
              m_vertexBuffer, m_stagingBuffer, m_bufferSize));

       if (updateBoundingSphere)
              calculateBoundingSphere(vertices);
}

VkResult updateClothStagingBuffer(std::vector<vh::vhVertex>& vertices,
       VkDeviceSize bufferSize, void* ptrToStageBufMem)
{
       memcpy(ptrToStageBufMem, vertices.data(), (size_t)bufferSize);

       return VK_SUCCESS;
}

VkResult updateClothVertexBuffer(VkDevice device, VmaAllocator allocator,
       VkQueue graphicsQueue, VkCommandPool commandPool, VkBuffer vertexBuffer,
       VkBuffer stagingBuffer, VkDeviceSize bufferSize)
{
       VHCHECKRESULT(vhBufCopyBuffer(device, graphicsQueue, commandPool, stagingBuffer,
              vertexBuffer, bufferSize));

       return VK_SUCCESS;
}
```

Code Block 6 Functions for updating vertices and copying updated vertex data into vertex buffer.

When destructed, the vertex staging buffer gets destroyed by the destructor of `VEClothMesh` and the vertex and index buffer by the parent destructor. The destructor of `VEMesh` was declared `virtual`, so that the buffers are destroyed correctly, even if the engine uses the concept of polymorphism.

```
VEMesh::~VEMesh()
{
	vmaDestroyBuffer(getEnginePointer()->getRenderer()->getVmaAllocator(),
		m_indexBuffer, m_indexBufferAllocation);
	vmaDestroyBuffer(getEnginePointer()->getRenderer()->getVmaAllocator(),
		m_vertexBuffer, m_vertexBufferAllocation);
}
```

Code Block 7 Destructor of mesh class.

```
VEClothMesh::~VEClothMesh()
{
	vmaUnmapMemory(getEnginePointer()->getRenderer()->getVmaAllocator(),
		m_stagingBufferAllocation);
	vmaDestroyBuffer(getEnginePointer()->getRenderer()->getVmaAllocator(),
		m_stagingBuffer, m_stagingBufferAllocation);
}
```

Code Block 8 Destructor of cloth mesh class.

3.1.2 Cloth Subrenderer

Only forward rendering is implemented for cloth. Cloth entities have got their own assigned subrenderer `VESubrenderFW_Cloth`. It works analogically to `VESubrenderFW_D` but loads a custom cloth fragment shader program. In the fragment shader code, there is a Boolean `drawShadow` which is true when the front and false if the back of a cloth is rendered. If it is false, no shadows are drawn onto the surface of the cloth, so that shadows are only drawn once on the cloth.

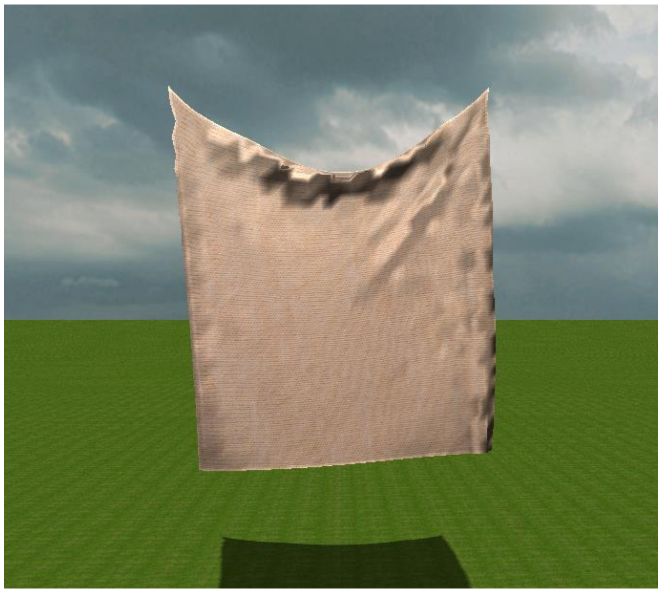


Image 5 Rendered cloth with shadows.

3.2 Cloth Simulation

3.2.1 Overview

The logic for the simulation is mainly implemented in the `Cloth`, `ClothMassPoint` and `ClothConstraint` class. They are members of the `VPEWorld` class which implements the physics engine. Additionally, there is a struct `ClothTriangle` which simply stores three indices of mass-points and is only used during the creation of a cloth for generating constraints. A member variable named `m_cloths` was added to `VPEWorld` analogically to `m_bodies`. Both are maps that store all instances of rigid or soft bodies as values with void pointers to their owners as their keys. Two loops were added to the `tick` function which iterates over all entries of `m_cloths`. One for integration and one for notifying the owners that the mesh has changed. The callback function that allows for doing so is defined in the main file (see section 3.5) and can be exchanged if a custom render engine is used or if different behavior is desired.

The following chapters deal with the concrete implementations of the concepts regarding simulation described in the Design chapter (see section 2.3).

3.2.2 Mass-Point Class

```
class ClothMassPoint
{
	public:
	std::vector<uint32_t> m_associatedVertices{};
	glmvec3 pos;
	glmvec3 prevPos;
	const glmvec3 initialPos;
	glmvec3 vel;
	real invMass;
	bool isFixed;
private:
	const real c_small = 0.01_real;
	const real c_verySmall = c_small / 50.0_real;
	const real c_collisionMargin = 0.045_real;
	const real c_friction = 300._real;
	const real c_damping = 0.1_real;

public:
	ClothMassPoint(glm::vec3 pos, bool isFixed = false) : pos{ pos }, prevPos{ pos },
		initialPos{ pos }, vel{ glmvec3(0._real) }, isFixed{ isFixed }, invMass{ 0 }
	{}

	void applyExternalForce(glmvec3 force, real dt) {…}

	void resolveGroundCollision(real dt) {…}

	void resolvePolytopeCollisions(const std::vector<std::shared_ptr<Body>>& bodies,
		real dt = 0._real) {…}

	void damp(real dt) {…}

private:
	void resolvePolytopeCollision(const std::shared_ptr<Body> body,
		glmvec3 massPointLocalPos, real dt) {…}
};
```

Code Block 9 Cloth mass-point class.

In addition to its position, previous position and velocity, a mass-point also stores its initial position, which is its position before the mesh of the cloth is changed. It is used for setting the transformation of a cloth. The associated vertices of a mass-point are all vertices of the mesh which have a corresponding initial position. Mass is implemented as inverted mass ($\frac{1}{mass}$) which simplifies the calculations of solving a constraint. Furthermore, a mass point can be fixed and possesses a few constants for damping, friction, and thresholding.

The constructor allows for setting the initial position and whether the point is fixed. The public functions are being called during a cloth's integration process where gravity and damping is applied, and collisions checks are initiated.

`resolvePolytopeCollisions` is called for all mass-points of a cloth every time-substep with a list of rigid bodies for which a collision is possible. For each rigid body the position of the mass-point is transformed into its local space.

```
glmvec3 massPointLocalPos = body->m_model_inv * glmvec4(pos, 1);
```

Code Block 10 Code transforming mass-point's position into rigid body's local space.

Also, a check is performed whether a given mass-point is within its bounding sphere. If so, `resolvePolytopeCollision` is called.

```
void resolvePolytopeCollision(const std::shared_ptr<Body> body, glmvec3 massPointLocalPos,
       real dt)
{
       std::pair<glmvec3, real> nearestProjectionPoint{ {}, INFINITY };

       for (const Face& face : body->m_polytope->m_faces)
       {
              real t = dot(face.m_face_vertex_ptrs[0]->m_positionL +
                     face.m_normalL * c_collisionMargin -
                     massPointLocalPos, face.m_normalL);

              if (t < c_small)
                     return;

              if (t < nearestProjectionPoint.second)
                     nearestProjectionPoint =
                            { massPointLocalPos + t * face.m_normalL, t };
       }


       prevPos = pos;
       pos = body->m_model * (glmvec4(nearestProjectionPoint.first, 1));
       vel -= vel * c_friction * dt;
}
```

Code Block 11 Function resolving possible collision between a mass-point and a rigid body.

The length `t` of the line segment between the transformed mass-point position and the spanned plane is calculated for every face of a rigid body. If `t` is negative, the mass-point must be outside of the rigid body and there cannot be a collision. If this is not the case for any of the faces of a rigid body, the position of the mass-point is moved onto the closest face by using the smallest value of `t`. Before setting the position, it is transformed back into global space. The velocity of the mass-points is changed according to the position change.

3.2.3 Constraint Class

`ClothConstraint` represents a distance constraint. Whether an instance is an edge or bending constraint is solely defined by which mass-points were used during its creation and is not relevant afterwards.

```
class ClothConstraint
{
public:
	ClothMassPoint* point0;
	ClothMassPoint* point1;
	real length;
	real compliance;
	const real c_threshold = 0.0001_real;


	ClothConstraint(ClothMassPoint* point0, ClothMassPoint* point1, real compliance)
		: point0{ point0 }, point1{ point1 }, compliance{ compliance }
	{
		length = glm::distance(point0->pos, point1->pos);
	}

	void solve(real dt) const {…}

};
```

Code Block 12 Cloth constraint class.

`solve` corrects the positions of the two associated mass-points towards where they are supposed to be according to the constraint. Towards each other if the distance in between them is greater than the rest-length and away from each other if smaller. The code of this function is fully based on the work of Macklin et al. [23]. The only original addition is that fixed points are not affected by constraints. If both mass-points of a constraint (in theory this constraint needs not to be created in the first place) are fixed, the function returns immediately.

```
void solve(real dt) const
{
	if (point0->isFixed && point1->isFixed)
		return;

	glmvec3 pos0 = point0->pos;
	glmvec3 pos1 = point1->pos;

	real lengthBetweenPoints = glm::distance(pos0, pos1);

	real lengthDifference = lengthBetweenPoints - length;

	if (fabs(lengthDifference) > c_threshold)
	{
		glmvec3 directionBetweenPoints = (pos1 - pos0) / lengthBetweenPoints;
		real lambda = -lengthDifference /
			(point0->invMass + point1->invMass + compliance / (dt * dt));

		glmvec3 correctionVec0 = -lambda * point0->invMass * directionBetweenPoints;
		glmvec3 correctionVec1 = lambda * point1->invMass * directionBetweenPoints;

		if (!point0->isFixed)
		{
			point0->pos += correctionVec0;
			point0->vel = (point0->pos - point0->prevPos) / dt;
		}

		if (!point1->isFixed)
		{
			point1->pos += correctionVec1;
			point1->vel = (point1->pos - point1->prevPos) / dt;
		}
	}
}
```

Code Block 13 Function solving a distance constraint.

3.2.4 Cloth Class

The callback functions for cloth are abstracted through type-aliases like the ones of rigid bodies.

```
using callback_move_cloth = std::function<void(double, std::shared_ptr<Cloth>)>;
using callback_erase_cloth = std::function<void(std::shared_ptr<Cloth>)>;
```

Code Block 14 Callback function type aliases.

The collections of mass-points, constraints, vertices, and nearby bodies are implemented as vectors. The fixed points, bending compliance (inverse of stiffness), amount of substeps and the amount by which unfixed points should be interpolated when transformations are applied can be set via the constructor. The constructor also initiates the generation of mass-points and constraints.

```cpp
class Cloth
{
public:
	std::string m_name;
	void* m_owner;
	callback_move_cloth m_on_move;
	callback_erase_cloth m_on_erase;
private:
	VPEWorld* m_physics;
	std::vector<ClothMassPoint> m_massPoints{};
	std::vector<ClothConstraint> m_constraints{};
	std::vector<vh::vhVertex> m_vertices;
	real m_maxMassPointDistance;
	const int c_substeps;
	const real c_movementSimulation;
	std::vector<std::shared_ptr<Body>> m_bodiesNearby;
	int_t m_gridX;
	int_t m_gridZ;
	int_t m_bodiesNearbyCount;

public:
	Cloth(VPEWorld* physics, std::string name, void* owner,
		callback_move_cloth on_move, callback_erase_cloth on_erase,
		std::vector<vh::vhVertex> vertices, std::vector<uint32_t> indices,
		std::vector<glmvec3> fixedPointsPositions, real bendingCompliance = 1,
		int substeps = 4, real movementSimulation = 0.8) {…}

	void integrate(const std::unordered_map<intpair_t, body_map>& rigidBodyGrid,
		double dt) {…}

	std::vector<vh::vhVertex> generateVertices() {…}

	void applyTransformation(glmmat4 transformation, bool simulateMovement) {…}

	void setTransformation(glmmat4 transformation, bool simulateMovement) {…}

private:
	void updateBodiesNearby(const std::unordered_map<intpair_t,
	body_map>& rigidBodyGrid) {…}

	void createMassPoints(const std::vector<vh::vhVertex>& vertices,
		const std::vector<glmvec3> fixedPointsPositions) {…}

	void calcMaxMassPointDistance() {…}

	std::vector<ClothTriangle> createTriangles(std::vector<uint32_t> indices) {…}

	void generateConstraints(const std::vector<ClothTriangle>& triangles,
		real bendingCompliance) {…}
};
```

Code Block 15 Cloth class.

The position of a cloth can be changed by applying transformations using `applyTransformation` or `setTransformation`. If cloth movement simulation is active, the positions of all mass-points that are not fixed are interpolated so that they get dragged along by the fixed points but do not fully rely on it to avoid stretching.

```
glmvec3 transformedPos = transformation * glmvec4(massPoint.pos, 1);
glmvec3 posToTransPos = transformedPos - massPoint.pos;
	massPoint.prevPos = massPoint.pos;
	massPoint.pos = massPoint.pos + posToTransPos * c_movementSimulation;
```

Code Block 16 Code moving mass-points by using interpolation if their position is not fixed.

`integrate` implements the algorithm outlined in the Design chapter (see section 2.3) which is fully based on the concepts presented by Macklin et al. [23].

```
void integrate(const std::unordered_map<intpair_t, body_map>& rigidBodyGrid, double dt)
{
	updateBodiesNearby(rigidBodyGrid);

	real rDt = dt / c_substeps;

	for (int i = 0; i < c_substeps; ++i)
	{
		for (ClothMassPoint& massPoint : m_massPoints)
		{
			massPoint.damp(rDt);
			massPoint.applyExternalForce(glmvec3{ 0, m_physics->c_gravity, 0 },
				rDt);
		}

		for (const ClothConstraint& constraint : m_constraints)
			constraint.solve(rDt);

			if (m_bodiesNearby.size())
				for (ClothMassPoint& massPoint : m_massPoints)
					massPoint.resolvePolytopeCollisions(m_bodiesNearby,
						rDt);

			if (m_massPoints[0].pos.y < m_maxMassPointDistance)
				for (ClothMassPoint& massPoint : m_massPoints)
					massPoint.resolveGroundCollision(rDt);
	}
}
```

Code Block 17 Function integrating cloth.

`updateBodiesNearby` implements broad-phase collision checking by checking which rigid bodies are within the cell or neighbor cells of the cloth. A cloth keeps track of its position within the broad-phase grid of the physics class. It does so by obtaining the size of the cells via its pointer to the physics class and storing its current cell position. After gravity has been applied and the constraints have been solved, narrow-phase collision checking is initiated by calling the mass-points' dedicated functions.

3.3 Performance Optimizations

3.3.1 Distance Function

Calculating the distance between two points in three-directional space is a relatively expensive calculation since it contains one square root operation. The cloth simulation uses distance constraints which need to be solved multiple times every timestep. This can cause a bottleneck and high computational costs. The Vienna Physics Engine uses OpenGL Mathematics [29] for vector and matrix calculations. This library offers a distance function that returns the distance between two vectors. By closely approximating this distance function without using a square root operation the cost could be reduced.

The first attempt to do so involved the infamous “Fast Inverse Square Root” function (also known as magic number approximation) which was famously used in the 1999 first-person shooter Quake III Arena. It causes a bit overflow on purpose for its calculation [30]. Therefore, it is dependent the nature of 32-bit architectures and can be seen as a hack. The decision not to use this approximation for the cloth simulation was made, since the code might not work on some architectures or if the data types used by the Vienna Physics Engine are modified.

The second approach involved the “Alpha Max Plus Beta Min” algorithm. This algorithm approximates the magnitude of a two-dimensional vector. The distance between two points can therefore be approximated by at first calculating the difference and then the magnitude approximation.

```
maxAbs = max(|x|, |y|)
minAbs = min(|x|, |y|)
magnitude = alpha * maxAbs + beta * minAbs
```

Code Block 18 Alpha Max Plus Beta Min algorithm.

Finding the optimum values for `alpha` and `beta` is an optimization problem which results in a maximum error smaller than 4% if `alpha ≈ 0.9604338701` and `beta ≈ 0.3978247347` are used [31]. Since cloth is simulated in three-dimensional space, this concept needs to be expanded. There are two ways to do so. Firstly, $\sqrt{x^2 + y^2 + z^2}$ can be rewritten as $\sqrt{\sqrt{x^2 + y^2}^2 + z^2}$. This allows for simply calling the “Min Max Plus Beta Min” algorithm twice to approximate the magnitude of a three-dimensional vector. It needs to be considered, that an error is introduced twice this way which adds up. An alternative approach is to expand this principle to an algorithm which at first determines the minimum, maximum and medium value and then uses three constants, alpha, beta, and gamma to approximate the magnitude. This optimization problem yields `alpha ≈ 0.9398086351`, `beta ≈ 0.3892814827` and `gamma ≈ 0.2987061876` [32].

```
maxAbs = max(|x|, |y|, |z|)
minAbs = min(|x|, |y|, |z|)
medAbs = max(min(|x|, |y|), min(max(|x|, |y|), |z|));
magnitude = alpha * maxAbs + beta * medAbs + gamma * minAbs
```

Code Block 19 Alpha Max Plus Beta Med Plus Gamma Min algorithm.

A simple program (see section 7) was written to compare the performance of the OpenGL Mathematics, “Alpha Max Plus Beta Min” and “Alpha Max Plus Beta Med Plus Gamma Min” algorithms and to calculate the mean error of the approximation. The program calculates the distance between pairs of pseudo-random three-dimensional vectors with every algorithm. The program was analyzed using Microsoft Visual Studio’s Performance Profiler and the results were as follows:

Table 1 Performance of distance approximation functions.

Algorithm	Relative CPU Usage	Mean Relative Error
OpenGL Mathematics Distance Function	30.84%	0.00%
Alpha Max Plus Beta Min	22.79%	4.15%
Alpha Max Plus Beta Med Plus Gamma Min	46.36%	2.87%

It must be noted that these results depend on the processor that is used. In this case it was an AMD Ryzen 5 5600x [33]. While the “Alpha Max Plus Beta Med Plus Gamma Min” algorithm can reduce the error that is caused when the original “Alpha Max Plus Beta Min” algorithm is called twice, it performs significantly worse than OpenGL Mathematics’ distance function and is therefore useless. The CPU used seems to be able to solve a square root operation faster on average than two min and two max function calls. The “Alpha Max Plus Beta Min” is in fact faster but introduces an average relative error of over 4%. This might not seem much but since the distance constraints of the cloth simulation are solved multiple times every timestep, the error adds up and grows too large. In praxis replacing OpenGL Mathematics’ distance function with the “Alpha Max Plus Beta Min” algorithm results in very fidgety cloth behavior and was therefore not implemented in the cloth simulation.

3.4 Deployment

The source code can be found on the GitHub repository [1] of the Vienna Physics Engine. This page also describes how to set up and run the application. For the render engine to work on Windows 10/11, the Vulkan SDK needs to be installed and an environment variable named “VULKAN_SDK” needs to exist which points to its installation directory. The physics engine can also be utilized in any C++ 20 project without the Vienna Vulkan Engine, so a custom render engine can be used for rendering if desired.

3.5 Explanation of Usage

The cloth simulation can be used by modifying the main.cpp file of a cloned Vienna Physics Engine repository according to this section. The callback functions that get called when a piece of cloth moves or is deleted in the simulation should already be defined. If not, they should be defined as follows.

```
inline VPEWorld::callback_move_cloth onMoveCloth =
	[&](double dt, std::shared_ptr<VPEWorld::Cloth> cloth)
{
	VEClothEntity* clothOwner = static_cast<VEClothEntity*>(cloth->m_owner);
	auto vertices = cloth->generateVertices();
	(static_cast<VEClothMesh*>(clothOwner->m_pMesh))->updateVertices(vertices);
};

inline VPEWorld::callback_erase_cloth onEraseCloth =
	[&](std::shared_ptr<VPEWorld::Cloth> cloth)
{
	VESceneNode* node = static_cast<VESceneNode*>(cloth->m_owner);
	getSceneManagerPointer()->deleteSceneNodeAndChildren(
		((VESceneNode*)cloth->m_owner)->getName());
};
```

Code Block 20 Cloth callback functions.

A new cloth entity can be created within the `loadLevel(uint32_t)` function of the user defined class derived from `VEEngine` or in an event by using the following code.

```
VESceneNode* pScene = getRoot();

VEClothEntity* clothEntity;
VECHECKPOINTER(clothEntity =
	getSceneManagerPointer()->loadClothModel(»name«, »path«, »filename«));

pScene->addChild(clothEntity);

auto vertices = ((VEClothMesh*) (clothEntity->m_pMesh))->getInitialVertices();
auto indices = ((VEClothMesh*) (clothEntity->m_pMesh))->getIndices();

std::vector<glm::vec3> fixedPoints = »vector of positions«;

auto physicsCloth = std::make_shared<VPEWorld::Cloth>(m_physics, »name«, clothEntity,
	onMoveCloth, onEraseCloth, vertices, indices, fixedPoints,
	»bending compliance«, »substeps«, »movement simulation constant«);

m_physics->addCloth(physicsCloth);
```

Code Block 21 Creating a new cloth instance.

The name of the cloth can be an arbitrary string. The path and filename need to identify the model obj file that should be used. The model must be infinitely thin, and the unmodified Vienna Vulkan Engine must be able to load it as a rigid body. The fixed points need to contain at least two vertex positions that are also part of the vertex list of the obj file for the cloth simulation to work. The position values need to match exactly. The line of code could for example look like this:

```
std::vector<glm::vec3> fixedPoints =
	{ {-1.000000, 2.000000, -0.000000}, {1.000000, 2.000000, 0.000000} };
```

Code Block 22 Example for defining fixed mass-points.

The bending compliance defines how stiff the piece of cloth will be. A compliance of zero corresponds to infinite stiffness and a large number to a small stiffness. The number of substeps specifies how many times the constraints will be solved per timestep. The movement simulation constant defines to which extent unfixed mass-points get transformed instead of dragged along when a transformation is applied with simulated movement. One means the unfixed points behave as if movement was not simulated and zero that they only get dragged along which results in the cloth being stretched.

Moving the cloth can be achieved by applying a transformation. The transformation needs to be expressed through a 4x4 matrix. It can be specified whether the transformation should be fully applied to all mass-points of the piece of cloth or only to the fixed points so that the others get dragged along which simulates real cloth movement.

```
VESceneNode* cloth = getSceneManagerPointer()->getSceneNode(»name«));

m_physics->getCloth(cloth)->applyTransformation(»transformation matrix«,
       »should simulate movement«);
```

Code Block 23 Applying a transformation to a cloth.

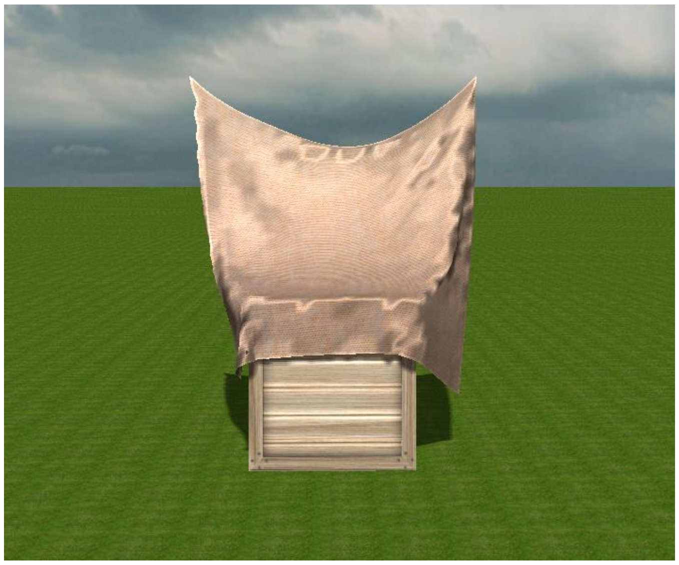


Image 6 A simulated piece of cloth colliding with a cube.

3.6 Known Issues

3.6.1 Rendering Related

The render process uses none of the advanced and sophisticated methods discussed in the literature analysis (see section 1.5) but only simple diffuse mapping which does not result in a realistic look. Additionally, cloth is only flat-shaded. This is because calculating smooth shading normals is computationally more expensive and the normals of a cloth mesh need to be recalculated every frame. The user could be given an option to enable smooth shading. Furthermore, the algorithm for calculating smooth shading normals can surely be implemented efficiently if the required data is structured adequately.

Extensively modifying the vector of vertices of a cloth every frame by appending a copy of every entry only to ensure that both sides cast shadows is a very inefficient solution.

3.6.2 Collision Related

There are multiple issues which occur when there is a collision between a piece of cloth and a rigid body. When a piece of cloth is being moved over a rigid body via transformations, it can stretch a lot if the rigid body has a surface perpendicular to the direction of the translation. This mainly happens because the position of a mass-point which penetrates a rigid body is corrected towards the direction of the normal of the closest face of the rigid body. If the normal of the face points into the opposite direction of the direction of the movement of the cloth this causes the piece of cloth to get stretched. A possible solution might be not to project the mass-point straight onto the face but to add a vector which points along the face towards the direction the mass-point should move.

If a piece of cloth is resting on a rigid body, its mass points are constantly in motion and not idle as they should be. The reason is that the process of applying gravity, solving constraints, and resolving collisions continues if cloth is lying on a rigid body. Elsewise, cloth could not slide across the surface of a rigid body.

There is a gap between cloth and rigid bodies even when they collide. The gap is created on purpose as a margin to avoid that the edges and corners of rigid bodies clip through the surface of the cloth. Using a more detailed cloth model with many vertices would allow for decreasing the gap.

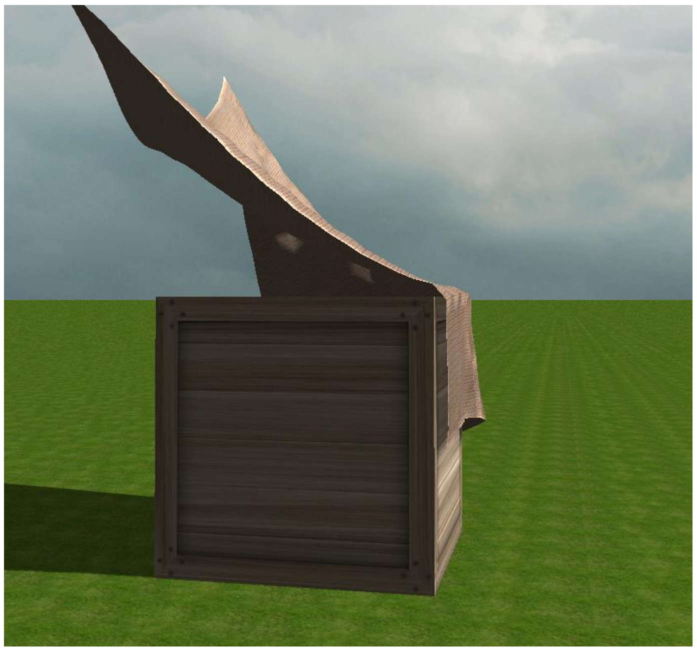


Image 7 Gap between cloth and rigid body.

If a transformation is applied which moves a piece of cloth up from a rigid body it was resting on, the cloth behaves in an unrealistic way. The mass-points which are not fixed do not remain resting on the rigid body's surface until they get dragged along but immediately get lifted by a distance that depends on the cloth's movement simulation constant. This is because of how unfixed mass-points get interpolated when a transformation is applied. To avoid this, an alternative method for preventing cloth to be stretched excessively when applying transformations needs to be found.

Collision detection and resolving is performed every substep. This is computationally very costly and possibly not necessary. Attempts to move the collision handling out of the substep loop did not lead to satisfactory results, however. Müller et al. propose a method to generate collision constraints outside of the substep loop [21]. With this approach the collision constraints still need to be solved every substep but at least the detection algorithm is only called once every timestep.

Neither self-collisions between different parts of a piece of cloth nor collisions between multiple pieces of cloth are being detected or resolved. This could be achieved by implementing the method described by Baraff et al. [27] mentioned in the literature analysis (see section 1.6).

3.6.3 Input Validation

In order to make the cloth simulation more user friendly, the arguments which are passed via the cloth constructor should be validated and adequate error messages or warnings displayed if they are not reasonable. For example, if a vertex position that should be fixed does not exists or if the loaded model is not appropriate for the cloth simulation.

4 Evaluation and Discussion

4.1 Methods

To check whether the goals of the software have been met, simulated cloth was visually compared with real cloth and performance testing has been done to confirm that the simulation satisfies the real-time requirement.

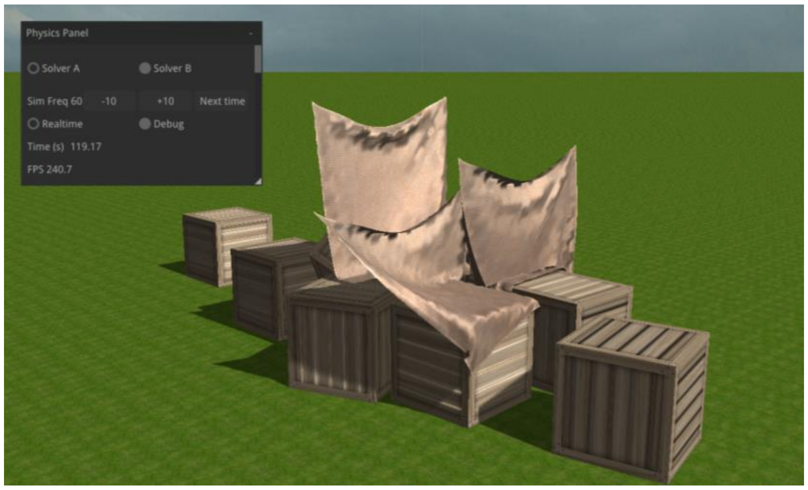
For performance testing the built-in FPS (frames per second) counter of the Vienna Physics Engine and the Performance Profiler of Microsoft Visual Studio have been used. The FPS counter estimates how many frames the software draws every second. The Performance Profiler allows for monitoring CPU, GPU, and memory usage.

It is difficult to validate that the explanation of usage and discussion of issues and possible improvements are sufficiently comprehensive and clear for students to act on it. Due to time constraints, it was not possible to let other students try to work with and adapt the cloth simulation.

4.2 Results And Discussion

A simulated simple piece of cloth behaves very similar to a real piece of cloth in the way it moves and interacts with rigid shapes.

An application runs in real-time if its responses to input is perceived to be immediate by the user [34] [35]. For videogames this is typically achieved if the framerate exceeds 30 FPS [36]. With three active pieces of cloth consisting of 1.089 vertices each and multiple colliding rigid bodies nearby, the application ran with an average of about 250 FPS during tests on an AMD Ryzen 5 5600x [33] processor and an NVIDIA GeForce 970 [37] graphics card.



```
Physics Panel
○ Solver A        ● Solver B
Sim Freq 60   -10   +10   Next time
○ Realtime        ● Debug
Time (s)  119.17
FPS 240.7
```

Image 8 Multiple cloths interacting with rigid bodies.

The performance of the application depends on how many vertices the cloth models that are being used consist of. The following graph shows the correlation between performance and the number of vertices of a single square-shaped piece of cloth without collisions. The specifications of the system are the same as above. The application was still able to stimulate a piece of cloth with 16.641 vertices (129x129) at real-time but froze when the model was subdivided once more.

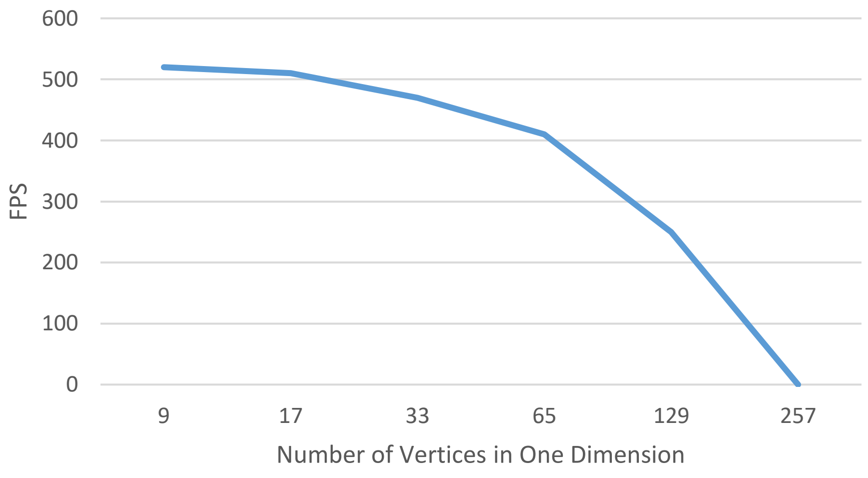


Image 9 Correlation between performance and number of vertices.

The first goal to enable users to simulate simple pieces of cloth by using the Vienna Physics Engine has been met since simulated cloth is perceived as cloth and the simulation runs in real-time.

Whether the second goal to enable students to use, improve and extend the simulation by explaining, analyzing, and discussing it in this work was met, cannot be conclusively determined at this point. Since this work attempts to cover all aspects of the solution, they should be, but only actual attempts of students to do so will tell.

4.3 Findings

Real cloth trends to have very detailed and complex surfaces. To imitate the look of real cloth, view dependent visual effects such as occlusion and self-shadows and illumination effects like specularity and interreflection need to be simulated. This requires more sophisticated methods than texture mappings as discussed in the literature review (see section 1.5).

Mass-spring systems are an intuitive approach to soft body simulations. Their biggest issue is their tendency to become unstable which requires the use of advanced integrations methods. (Extended) Position Based Dynamics [21] [23] do not pose this problem and also allow for a simple way of resolving collisions. Furthermore, this method can potentially also handle collisions between different cloths and self-collision of cloth when combined with other methods (see section 1.6).

When the design process of the cloth simulation was started, it was not yet clear that the render engine would need to be modified in addition to the physics engine to allow for rendering cloth. It then also became apparent, that the render engine was not designed with the concept of rendering soft bodies in mind. This posed several challenges. Modifying the already existing code of the engine infringes backwards compatibility for applications that have been created using the engine, while achieving the desired functionality solely through new classes and additional code leads to higher complexity and poor comprehensibility.

The “Alpha Max Plus Beta Min” algorithm – which approximates the magnitude of a two-dimensional vector - can be extended for approximating the distance function of three-dimensional vectors. While a mean average error of under 3% was achieved, it was still too inaccurate to be used for solving distance constraints. Additionally, depending on the implementation, the performance was only slightly better than OpenGL Mathematics’ distance function or even worse.

5 Conclusion and Future Work

5.1 Summary of Findings

Rendering cloth in a photorealistic manner is not a trivial task.

Using (Extended) Position Based Dynamics [21] [23] is a suitable approach to cloth simulations which offers many advantages especially in terms of stability compared to other methods like mass-spring systems.

Extending a software that was not designed with these specific extensions in mind poses unexpected challenges. To keep the code of the extension logical and simple, the underlaying code of the software must most likely be modified. An alternative approach is to accept a higher complexity on the side of the extension for the sake of keeping the base code mainly unchanged.

The “Alpha Max Plus Beta Min” algorithm can be used for a fast and good approximation of the distance function. Solving distance constraints using Position Based Dynamics however requires more accurate results since the error adds up.

5.2 Limitations of the Approach

Engineering an extension for a software that was not designed with the requirements of this extension in mind comes with a unique set of challenges. Major modifications of the pre-existing code of the Vienna Physics Engine were avoided which sometimes led to less efficient design and implementation decisions.

The available time frame for this work did not allow for implementing both a cloth simulation and realistic cloth rendering. Therefore, none of the sophisticated methods for rendering cloth presented in the literature analysis were implemented.

5.3 Future Work

All presented issues of the implementation (see section 3.6) could be further analyzed and solved. They can be summarized into two core aspects. The first one comprises improvements of the visual output of the rendering process. The second aspect concerns collision handling. The way collisions between cloth and rigid bodies are resolved could be improved while self-collision and collisions between different pieces of cloth could be implemented.

Furthermore, it might be possible to improve the performance of the simulation by using constraints different from distance constraints to simulate cloth or by finding a suitable way to approximate the distance function (see section 3.3.1).

The cloth simulation could be used as a foundation to create a clothing simulation for videogame-characters. Also, since Extended Position Based Dynamics can be used to simulate soft bodies in general, the system could be expanded so that it can simulate other types of soft bodies as well.

The mesh and cloth mesh class of the Vienna Vulkan Engine could be hierarchically restructured and refactored so that there are three classes. Firstly, the default mesh class whose vertices cannot change during runtime. Secondly, a mutable mesh class that inherits from the default mesh class but allows for updating its vertices. This mesh can be useful for various further extensions like other soft body simulations or animations for example. Thirdly, the cloth mesh which would then inherit from the mutable mesh class and implement all the cloth-specific functionality.

Cloth can currently only be rendered through forward rendering. Support for other methods such as deferred rendering or raytracing could be implemented.

6 References

[1] H. Hlavacs, “The Vienna Physics Engine (VPE) Repository,” [Online]. Available: https://github.com/hlavacs/ViennaPhysicsEngine. [Accessed 01 07 2023].

[2] H. Hlavacs, “The Vienna Vulkan Engine (VVE) Repository,” [Online]. Available: https://github.com/hlavacs/ViennaVulkanEngine. [Accessed 01 07 2023].

[3] The Khronos Group Inc., “VkBuffer(3) Manual Page,” 30 06 2023. [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkBuffer.html. [Accessed 07 03 2023].

[4] A. Overvoorde, “Index Buffer,” [Online]. Available: https://vulkan-tutorial.com/Vertex_buffers/Index_buffer. [Accessed 03 07 2023].

[5] E. Diamond, “Creative Technologies Project: Implementing soft body dynamics and collisions in Vulkan,” 2020.

[6] The Khronos Group Inc, “VkBufferUsageFlagBits(3) Manual Page,” 30 06 2023. [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkBufferUsageFlagBits.html. [Accessed 04 07 2023].

[7] A. Overvoorde, “Staging buffer,” [Online]. Available: https://vulkan-tutorial.com/Vertex_buffers/Staging_buffer. [Accessed 04 07 2023].

[8] The Khronos Group Inc, “VkMemoryPropertyFlagBits(3) Manual Page,” 30 06 2023. [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkMemoryPropertyFlagBits.html. [Accessed 04 07 2023].

[9] Advanced Micro Devices, Inc., “Vulkan® Memory Allocator,” 2023. [Online]. Available: https://gpuopen.com/vulkan-memory-allocator/. [Accessed 04 07 2023].

[10] Advanced Micro Devices, Inc., “Memory allocation,” 2022. [Online]. Available: https://gpuopen-librariesandsdks.github.io/VulkanMemoryAllocator/html/index.html. [Accessed 04 07 2023].

[11] K. Wu and C. Yuksel, “Real-time Cloth Rendering with Fiber-level Detail,” *IEEE Transactions on Visualization and Computer Graphics,* 26 07 2017.

[12] S. Zhao, W. Jakob, S. Marschner and K. Bala, “Building Volumetric Appearance Models of Fabric Using Micro CT Imaging,” *ACM Transactions on Graphics,* vol. 30, no. 4, 07 2011.

[13] S. Zhao, W. Jakob, S. Marschner and K. Bala, “Structure-Aware Synthesis for Predictive Woven Fabric Appearance,” *ACM Transactions on Graphics,* vol. 31, no. 4, p. 10, 07 2012.

[14] P. Khungurn, D. Schroeder, S. Zhao, K. Bala and S. Marschner, “Matching Real Fabrics with Micro-Appearance Models,” *ACM Transactions on Graphics,* vol. 35, no. 1, p. 26, 12 2015.

[15] K. Daubert, H. P. A. Lensch, W. Heidrich and H.-P. Seidel, “Efficient Cloth Modeling and Rendering,” *Rendering Techniques 2001,* pp. 63-70, 2001.

[16] M. Sattler, R. Sarlette and R. Klein, “Efficient and Realistic Visualization of Cloth,” *Eurographics Workshop on Rendering,* 2003.

[17] T. Stuyck, Cloth Simulation for Computer Graphics, 1 ed., Springer Cham, 2018.

[18] J. Weil, “The synthesis of cloth objects,” *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques,* p. 49–54, 08 1986.

[19] J. E. Chadwick, D. R. Haumann and R. E. Parent, “Layered Construction for Deformable Animated Characters,” vol. 23, no. 3, p. 243–252, 07 1989.

[20] J. Meist, R. Guha and S. Chaudhry, “3D Soft Body Simulation Using Mass-spring System with Internal Pressure Force and Simplified Implicit Integration,” *Journal of Computers,* vol. 2, no. 8, 10 2007.

[21] M. Müller, B. Heidelberger, M. Hennix and R. John, “Position based dynamics,” *Journal of Visual Communication and Image Representation,* vol. 18, no. 2, pp. 109-118, 2007.

[22] A. Nealen, M. Mueller, R. Keiser, E. Boxerman and M. Carlson, “Physically Based Deformable Models in Computer Graphics,” *Computer Graphics Forum,* vol. 25, no. 4, pp. 809-836, 2006.

[23] M. Macklin, M. Müller and N. Chentanez, “XPBD: position-based simulation of compliant constrained dynamics,” *Proceedings of the 9th International Conference on Motion in Games,* p. 49–54, 2016.

[24] F. Fazioli, F. Ficuciello, G. A. Fontanelli, B. Siciliano and L. Villani, “Implementation of a soft-rigid collision detection algorithm in an open-source engine for surgical realistic simulation,” *2016 IEEE International Conference on Robotics and Biomimetics (ROBIO),* pp. 2204-2208, 2016.

[25] T. Möller and B. Trumbore, “Fast, Minimum Storage Ray/Triangle Intersection,” *ACM SIGGRAPH 2005 Courses,* p. 7–es, 2005.

[26] J. Fan, Q. Wang, S.-F. Chen, M. M. F. Yuen and C. C. Chan, “A spring–mass model-based approach for warping cloth patterns on 3D objects,” *The Journal of Visualization and Computer Animation,* vol. 9, no. 4, pp. 215-227, 1998.

[27] D. Baraff, A. Witkin and M. Kass, “Untangling Cloth,” *ACM Transactions on Graphics,* vol. 22, no. 3, p. 862–870, 07 2003.

[28] C. Ericson, Real-Time Collision Detection (The Morgan Kaufmann Series In Inteacive 3-D Technolagy), CRC Press, 2004, 2004.

[29] G-Truc Creation, “OpenGL Mathematics (GLM),” [Online]. Available: https://github.com/g-truc/glm. [Accessed 11 07 2023].

[30] Rys for Software, “Origin of Quake3's Fast InvSqrt() - Page 1,” 29 11 2006. [Online]. Available: https://www.beyond3d.com/content/articles/8/. [Accessed 11 07 2023].

[31] Iowegian International Corporation; Griffin, Grant, “DSP Trick: Magnitude Estimator,” 02 10 1999. [Online]. Available: http://dspguru.com/dsp/tricks/magnitude-estimator/. [Accessed 11 07 2023].

[32] uranix (https://math.stackexchange.com/users/200750/uranix), “Alpha max plus beta min algorithm for three numbers,” 14 05 2015. [Online]. Available: https://math.stackexchange.com/q/1282495. [Accessed 11 07 2023].

[33] Advanced Micro Devices, Inc., “AMD Ryzen 5 5600X Desktop-Prozessoren,” [Online]. Available: https://www.amd.com/de/products/cpu/amd-ryzen-5-5600x. [Accessed 10 07 2023].

[34] SUSE, “Definition Real-Time,” [Online]. Available: https://www.suse.com/suse-defines/definition/real-time. [Accessed 10 07 2023].

[35] B. Lutkevich, “real-time application (RTA),” 05 2022. [Online]. Available: https://www.techtarget.com/searchunifiedcommunications/definition/real-time-application-RTA. [Accessed 10 07 2023].

[36] M. Klappenbach, “Understanding and Optimizing Video Game Frame Rates,” Dotdash Meredith, 5 11 2022. [Online]. Available: https://www.lifewire.com/optimizing-video-game-frame-rates-811784. [Accessed 10 07 2023].

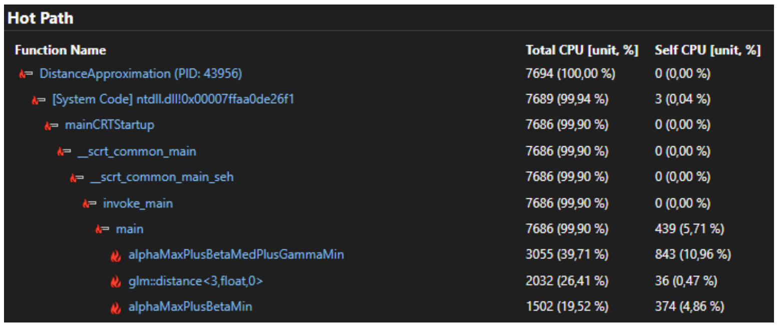
[37] NVIDIA Corporation, “SERIE GEFORCE GTX 900,” [Online]. Available: https://www.nvidia.com/en-us/geforce/900-series/. [Accessed 10 07 2023].

7 Appendix

This is the function that was used for analysing the “Alpha Max Plus Beta Min” algorithms. `vecPositions` was populated with many pairs of pseudo-random three-dimensional vectors.

```
std::vector<std::pair<glm::vec3, glm::vec3>> vecPositions = {
	{ {0.1, 2, 3}, {0.3, 4, 0} },
	…
};

int main()
{
	float avgErrorAlphaBeta = 0;
	float avgErrorAlphaBetaGamma = 0;

	for (const std::pair<glm::vec3, glm::vec3>& positions : vecPositions)
	{
		glm::vec3 pos0 = positions.first;
		glm::vec3 pos1 = positions.second;

		float realDistance = glm::distance(pos0, pos1);

		float distanceAlphaBeta =
			alphaMaxPlusBetaMin(alphaMaxPlusBetaMin(pos0.x - pos1.x,
				pos0.y - pos1.y), pos0.z - pos1.z);

		float distanceAlphaBetaGamma =
			alphaMaxPlusBetaMedPlusGammaMin(pos0.x - pos1.x, pos0.y - pos1.y,
				pos0.z - pos1.z);

		avgErrorAlphaBeta +=
			std::abs((realDistance - distanceAlphaBeta) / realDistance);
		avgErrorAlphaBetaGamma +=
			std::abs((realDistance - distanceAlphaBetaGamma) / realDistance);
	}

	avgErrorAlphaBeta = avgErrorAlphaBeta / vecPositions.size();
	avgErrorAlphaBetaGamma = avgErrorAlphaBetaGamma / vecPositions.size();

	std::cout << "Average Error Alpha Beta: " << avgErrorAlphaBeta << std::endl;
	std::cout << "Average Error Alpha Beta Gamma: " << avgErrorAlphaBetaGamma <<
		std::endl;

	return 0;
}
```

The findings are based on the output and the results of Microsoft Visual Studio’s performance profiler.



Hot Path

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
DistanceApproximation (PID: 43956)	7694 (100,00 %)	0 (0,00 %)
[System Code] ntdll.dll!0x00007ffaa0de26f1	7689 (99,94 %)	3 (0,04 %)
mainCRTStartup	7686 (99,90 %)	0 (0,00 %)
__scrt_common_main	7686 (99,90 %)	0 (0,00 %)
__scrt_common_main_seh	7686 (99,90 %)	0 (0,00 %)
invoke_main	7686 (99,90 %)	0 (0,00 %)
main	7686 (99,90 %)	439 (5,71 %)
alphaMaxPlusBetaMedPlusGammaMin	3055 (39,71 %)	843 (10,96 %)
glm::distance<3,float,0>	2032 (26,41 %)	36 (0,47 %)
alphaMaxPlusBetaMin	1502 (19,52 %)	374 (4,86 %)