Abstract

Deformation of Soft Bodies on GPU

This thesis provides an overview of real-time soft body deformation methods. The goal of this study is to design and implement numerical library solving the problem of physically correct soft body deformations. GPU accelerated Finite Element Method is used which solves the problem of solid mechanics for linear geometry and linear material properties. The library is integrated into existing open source game engine.

Keywords: Soft Body, Finite Element Method, Solid Mechanics, Real-Time, GPU, CUDA, Physics engine, Game physics, Simulation, Deformation

Streszczenie

Deformacje Ciał Odkształcalnych na GPU

Słowa kluczowe: Deformacje Ciał Odkształcalnych, Metoda Elementów Skończonych, Mechanika Ciał Stałych ,GPU, CUDA

Jakub Ciecierski
Nr albumu 243260

Warsaw, ...

Declaration

I hereby declare that the thesis entitled „Deformation of Soft Bodies on GPU", submitted for the magisters degree, supervised by dr inż. Joanna Porter-Sobieraj, is entirely my original work apart from the recognized reference.

.....

Jakub Ciecierski

Spis treści

Introduction

With the growth of popularity in virtual reality hardware comes great responsibility for software engineers to provide realistic and detailed experiences. User wearing virtual reality headset such as Oculus Rift is closer to the virtual environment than ever before. She will expect full interaction with the environment around her. The object she will pick up can be seen from very close distance. This requirement pushes software and hardware engineers alike to the frontier of computer graphics to provide detailed and authentic experiences. However, the user will also expect the virtual object to behave in a physically correct way. Virtual reality controllers gives us a chance to grab and interact with objects using our own hands. We should expect a virtual object to deform when enough pressure is applied to it. Thus, it is our responsibility as engineers to develop physics engines suitable for such tasks.

Real-time soft body deformation simulations have many practical applications. It provides more realistic and thus more fun gaming experiences. Soft body simulation is extremely useful in educational applications such as real time virtual surgery tutorial. Real time virtual car crash testing can provide fast and extremely cheap way to analyse car deformation in case of accident.

Topic of this thesis is motivated by the ever increasing requirements of virtual environment users and by the rapid growth of hardware capabilities in a form of GPU computational power. This thesis provides an overview of current methods used in real-time soft body deformations. The goal is to design and implement numerical library that would solve the problem of soft body deformation in real-time. Moreover, the library should provide friendly interface which support easy integration into existing virtual reality engines(e.g. game engines). The author proposes an integration into open source engine of his own creation [22].

The author chooses Finite Element Method(FEM) based simulation of soft bodies. It is the most physically correct method studied in this thesis. However, it comes with a price of expensive computations. FEM is used to solve the problem of solid mechanics with linear geometry and linear material properties. The linearity of the system enables us to work only with linear system of equations which can be computed much faster than the non-linear systems. However, the linear formulation is visibly pleasing only for small deformations. Linear tetrahedron finite element

is chosen which is quite suitable for real time simulations. Implicit euler method is used for integrating the differential equations of the dynamic system.

Understanding FEM from theoretical point of view might seem daunting at first. However, the study of applied mechanics have been extremely active for over three decades. Zienkiewicz in [2] and [3] provides thorough explanation of FEM and its applications in solid mechanics. Kleiber and Kowalczyk [1] familiarise the reader with the basics of tensor algebra, stress analysis and material properties. Lectures at University of Colorado [5] guide the reader through linear FEM. Moreover, they provide example functions written in Mathematica that can be used to test implementation of custom FEM solver. Altair University [4] shows practical applications of FEM aimed at material engineers.

Parker [11] implements linear FEM solver that focuses on game environments settings. He designed an algorithm based on implicit time integration on the CPU side. Moreover, the solution contains a fracture mechanics which enables objects to be broken under certain stress threshold. Faure et al. in [12] provide alternative to classical FEM called Tensor-Mass approach. This approach computes the forces applied directly on the vertices. Faure et al. implements a GPU accelerated solver using the SOFA framework [16]. Zhuang et al. were interested in non-linear FEM solver [13]. Experiments of their CPU implementation gave reasonable results thanks to the diagonalization of mass and damping matrices.

In chapter 1 the thesis starts with a background study of current methods of solving the problem of soft body deformations. It familiarises reader with numerical integration of differentiable equations and provides an overview of currently commercially available physics engines. Further in chapters 2 and 3 detailed explanation of solid mechanics and finite element method is provided. Here, the reader is set on a journey to transform linear partial differential equations(PDE) to linear system of algebraic equations. Then in chapter 4 method of solving the system of linear equations is described in a form of Conjugate Gradient. One of the greatest difficulties in FEM solver is creating an 3D computation mesh satisfying our problem domain. This thesis is out of scope of detailed explanations of tetrahedralization process, however a short overview is provided in chapter 5. Chapter 6 shows the implementation of the library and its integration into open source game engine. The thesis ends with chapters 7 and 8 where the author tests and provides conclusions of created algorithm.

Background

Mass Spring

Formulation

The simplest method to simulate soft body deformation is Mass Spring System. Such a system includes a set of N particles with masses m_i, positions x_i and velocities v_i, where $i \in 1...N$. The particles are connected by a set of springs S. A single spring $s \in S$ consists of $s = (i, j, l_0, k_s, k_d)$, where i and j are the indices of connected particles, l_0 is the rest length, k_s is the spring stiffness and k_d is the damping coefficient. To calculate the forces acting on particles i and j, we use the following formula:

$$f_i^S = f^S(x_i, x_j) = k_s \frac{x_j - x_i}{|x_j - x_i|}(|x_j - x_i| - l_0) \tag{1.1}$$

$$f_j^S = f^S(x_j, x_i) = -f^S(x_i, x_j) \tag{1.2}$$

It is easy to see that the forces conserve momentum, since $(f_i + f_j = 0)$

We apply damping by computing the damping forces:

$$f_i^D = f^D(x_i, v_i, x_j, v_j) = k_d(v_j - v_i)\frac{x_j - x_i}{|x_j - x_i|} \tag{1.3}$$

$$f_j^D = f^D(x_j, v_j, x_i, v_i) = -f_i^D \tag{1.4}$$

Combining two forces we get the final spring force

$$f(x_i, v_i, x_j, v_j) = f^S(x_i, x_j) + f^D(x_i, v_i, x_j, v_j) \tag{1.5}$$

Simulation

In order to simulate the mass spring system, we use the Newton's second law of motion,

$$f = m\ddot{x} \tag{1.6}$$

where f is the force, m is the mass and $\ddot{x}$ is the acceleration or the second derivative of position with respect to time. By transforming the equation to solve for acceleration, we get a second order ordinary differential equation(ODE):

$$\ddot{x} = \frac{f}{m} \tag{1.7}$$

In order to solve it, we can split this equation into two first order ODEs

$$\dot{v} = \frac{f}{m} \tag{1.8}$$

$$\dot{x} = v \tag{1.9}$$

Analytically, these can be solved by definite integrals:

$$v(t) = v_0 \int_{t_0}^{t} \frac{f(t)}{m} dt \tag{1.10}$$

$$x(t) = x_0 \int_{t_0}^{t} v(t) dt \tag{1.11}$$

where $v_0 = v(t_0)$ and $x_0 = x(t_0)$ are the initial conditions.

Numerical Integration

Explicit Euler Integration

One of the most basic numerical integration of ODE is explicit Euler integration scheme. The scheme approximates the derivatives using finite differences:

$$\dot{v} = \frac{v^{t+1} - v^t}{\Delta t} \tag{1.12}$$

$$\dot{x} = \frac{x^{t+1} - x^t}{\Delta t} \tag{1.13}$$

where Δt is a discrete time step and t is the index of the simulation iteration. By substituting these equations into Eq. 1.8 and Eq. 1.9, we get get the explicit Euler integration method:

$$v^{t+1} = v^t + \Delta t \frac{f(x^t, v^t)}{m} \tag{1.14}$$

$$x^{t+1} = x^t + \Delta t v^{t+1} \tag{1.15}$$

The term 'explicit' comes from the fact that information of the next time step can be directly computed using the information at the current time step.

The entire simulation can be summed with the following algorithm:

f^g is the gravity force and f^{coll} collision forces.

Algorithm 1 Mass Spring Simulation

1: **procedure** Simulation
2: **while** true **do**
3: **for** all particles i **do**
4: $f_i = f^g + f_i^{coll} + \sum_{j,(i,j)\in S} f(x_i, v_i, x_j, v_j)$
5: **for** all particles i **do**
6: $v_i = v_i + \Delta t \frac{f_i}{m_i}$
7: $x_i = x_i + \Delta t v_i$

A known drawback for explicit Euler integration is the fact that it requires small time steps to remain stable. This problem occurs because explicit Euler does not account for the near future and it assumes that the force is constant during the entire time step. Let us assume a system of two particles connected with a spring. Assume the following configuration: the spring is stretched and the two particles start moving towards each other. If we take a large time step to compute the next configuration, the particles might pass the equilibrium configuration, which in theory means that the force should change its sign during that time step. Sadly, since the force is constant throughout the entire time step, the sign change of the force is not accounted for. This might lead to particles overshooting and gaining energy which in turn leads to a so called simulation explosion. Other numerical integration methods exist that are more accurate. Among the most popular are the second and fourth order Runge-Kutta integrators. These schemes compute forces multiple times during a single time step, which might reduce the effect of the problem mentioned above.

Runge-Kutta Integration

The second order Runge-Kutta integrator has a different method of numerically solving ODEs. The approximation of explicit Euler Eq. 1.14 and Eq. 1.15 are instead computed by the formulas:

$$\begin{aligned} a_1 &= v^t \\ a_2 &= \frac{f(x^t, v^t)}{m} \\ b_1 &= v^t + \frac{\Delta t}{2} a_2 \\ b_2 &= \frac{f(x^t + \frac{\Delta t}{2} a_1, v^t + \frac{\Delta t}{2} a_2)}{m} \\ x^{t+1} &= x^t + \Delta t b_1 \\ v^{t+1} &= v^t + \Delta t b_2 \end{aligned} \tag{1.16}$$

It is easy to see that the forces are computed twice during one time step. This makes the second order Runge-Kutta integrator more accurate compared to the simple first order explicit Euler method.

One of the most popular methods of integrating ODE is a forth order Runge-Kutta integrator. It extends the the second order Runge-Kutta by computing the force four times during a single time step. Making it even more accurate. The accuracy obviously comes with longer computations.

Implicit Euler

Another way to improve stability is to use implicit integrator. Among the most popular is the implicit Euler method. As opposed to explicit integrators, the implicit is more physically correct.

$$\begin{aligned} v^{t+1} &= v^t + \Delta t \frac{f(x^{t+1})}{m} \\ x^{t+1} &= x^t + \Delta t v^{t+1} \end{aligned} \tag{1.17}$$

First difference lies in the force function f. Now it only depends on the position. In another words, the force does not include friction. It is said that implicit integration introduces enough numerical damping to accommodate for physical damping. However, if needed, the friction force can be added in the explicit step after the implicit solve. The most important change, is the fact that the force f depends now on the position of next step x^{t+1}. Thus, it is no longer possible to explicitly compute the two equations. Instead we now deal with a algebraic system, with unknowns being x^{t+1} and v^{t+1}.

In order to compute these equation we must first construct the algebraic system. The position, velocities and forces are concatenated into vectors:

$$\begin{aligned} x &= [x_1, x_2, ..., x_n] \\ v &= [v_1, v_2, ..., v_n] \\ f(x) &= [f_1(x_1, ..., x_n), ..., f_n(x_1, ..., x_n)] \end{aligned} \tag{1.18}$$

Further, we construct a mass matrix $M \in \mathbb{R}^{3N \times 3N}$ which is diagonal with values $m_1, m_1, m_1, ..., m_N, m_N, m_N$ on the diagonal.

$$Mv^{t+1} = Mv^t + \Delta t f(x_{t+1}) \tag{1.19}$$

$$x^{t+1} = x + \Delta t v^{t+1} \tag{1.20}$$

Substituting Eq. 1.20 into Eq. 1.19, results in single system of algebraic equations:

$$Mv^{t+1} = Mv^t + \Delta t f(x + \Delta t v^{t+1}) \tag{1.21}$$

We solve this system for v^{t+1}.

Example implementation of mass spring system can found in open source software created by the author of this thesis in [24].

Conclusions

Mass spring systems are easy to implement and for many applications give good enough results(e.g. computer games). However, relatively expensive ODE integrators have to be used in order to keep the simulation stable. Moreover, modelling physically correct materials can be a complicated task, since the parameters of the system hardly reflect reality.

Position Based Dynamics

As the name suggests, Position Based Dynamics(PBD) omits integrating over velocity and works directly on positions. The biggest advantage over mass spring system is avoidance of overshooting problem during integration step.

Formulation

The system of PBD includes a set of N particles and a set of M constraints. Each particle i has three attributes:

1. mass m_i
2. position x_i
3. velocity v_i

Each constraint j has five attributes:

1. Cardinality - n_j
2. Scalar constraint function - $C_j : \mathbb{R}^{3n_j} \rightarrow \mathbb{R}$
3. Set of indices - $\{i_1, ..., i_{n_j}\}, i_k \in [1, ..., N]$
4. Stiffness parameter - $k_j \in [0...1]$
5. Type - *unilateral* or *bilateral*

Algorithm 2 Particle Based Dynamics

1: **procedure** Simulation
2: **while** true **do**
3: **for** all particles i **do**
4: $v_i = v_i + \Delta t \frac{f_i}{m_i}$
5: $p_i = x_i + \Delta t v_i$
6: $generateCollisionContraints(x_i, p_i)$
7: **while** iteratively **do**
8: $projectContraints(C_1, ..., C_{M+M_{ext}}, p_1, ..., p_N)$
9: **for** all particles i **do**
10: $v_i = \frac{(p_i - x_i)}{\Delta t}$
11: $x_i = p_i$

It is said that bilateral constraint j is satisfied if $C_j(x_{i_1}, ..., x_{i_{n_j}}) = 0$ or if the case of unilateral $C_j(x_{i_1}, ..., x_{i_{n_j}}) \geqslant 0$. The strength of the constraint is defined by the stiffness parameter k_j.

Given initial conditions for positions and velocities the simulation proceeds as follows:

The lines 4 and 5 compute explicit Euler integration on velocities and positions. However, output positions p_i are only used as predictions. The line 6 generates external constrain such as collisions. The original and predicted positions x_i, p_i can be used in this step in order to perform continuous collision detection. The simulation then computes line 8 which iteratively corrects the predicted positions such that they satisfy the M_{ext} external and M internal constraints. Finally, the corrected positions p_i are used to compute velocities and positions. For more information about PDB the reader is encouraged to read [6]. PDB has found its usage in nvidia's Flex library [10].

Finite Element Method

Mass spring systems can not simulate physically correct volumetric effects. Moreover, it highly depends on the structure of the spring mesh. Finally, the parameters in previous soft body dynamics systems have no physical intuition. Thus it is hard to create different physical materials. All these limitations can be suppressed by the Finite Element Method. Soft body dynamics based on FEM are not yet common in the real time environment. One of the most successful usages of FEM in real time simulation were done by [11]. The authors were able to integrate their FEM

solver in the game called Star Wars: The Force Unleashed. The details on FEM formulation will follow in next chapters.

Solid Mechanics - Part 1

Formulation

Displacement

Displacement field $u(x)$ is a first order tensor, or a three dimensional vector. It defines the translation of body object. For clarity we will often omit the input position vector x and use the simplified tensor notation u_i

Strain

Strain tensor is the measure of elongation of the material. It is defined as second order symmetric tensor ϵ.

$$\epsilon = \begin{bmatrix} \epsilon_{11} & \epsilon_{12} & \epsilon_{13} \\ \epsilon_{12} & \epsilon_{22} & \epsilon_{23} \\ \epsilon_{13} & \epsilon_{23} & \epsilon_{33} \end{bmatrix} \tag{2.1}$$

Strain tensor can be computed in many ways. The linear version ϵ_C called Cauchy's linear strain tensor is defined by:

$$\epsilon_C = \frac{1}{2}(\Delta u + (\Delta u)^T) \tag{2.2}$$

where the gradient of displacement u is defined:

$$\Delta u = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} \\ u_{2,1} & u_{2,2} & u_{2,3} \\ u_{3,1} & u_{2,2} & u_{3,3} \end{bmatrix} \tag{2.3}$$

In tensor notation the spatial derivative is denoted by: $u_{1,1} = \frac{\partial u_1}{\partial x_1}$.

Stress

Let us defined traction force vector t as a density of forces f acting on an area A of a certain body point:

$$t = \frac{df}{dA} \tag{2.4}$$

Traction force acts with some magnitude along the normal vector n to the snippet element with area A. The magnitude is defined by second order, symmetric stress tensor σ.

$$t = \sigma n \tag{2.5}$$

Stress tensor is defined by:

$$\sigma = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{12} & \sigma_{22} & \sigma_{23} \\ \sigma_{13} & \sigma_{23} & \sigma_{33} \end{bmatrix} \tag{2.6}$$

Constitutive Laws

We defined three dimensional Hooke's law which defines a linear relation between stress and strain

$$\sigma_{ij} = C_{ijkl}\epsilon_{kl} \tag{2.7}$$

The fourth order tensor C is called a spring stiffness tensor. Tensor C does not depend on x and is composed of only constant material parameters. In case of isotropic materials(material that behave the same in all directions of motion) the Hooke's law can be defined by the system:

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{31} \end{bmatrix} = \frac{E}{(1+nu)(1-2nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-2\nu & 0 & 0 \\ 0 & 0 & 0 & 0 & 1-2\nu & 0 \\ 0 & 0 & 0 & 0 & 0 & 1-2\nu \end{bmatrix} \begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ \epsilon_{12} \\ \epsilon_{23} \\ \epsilon_{31} \end{bmatrix} \tag{2.8}$$

where E is Young's modulus that describes the elastic stiffness, ν is the Poisson's coefficient that describes the conversation of volume within the material.

Strong formulation

A strong formulation can be described by the following set of partial equations(PDE):

$$\begin{aligned} \sigma_{ij,j} + \hat{f}_i = 0 \quad & x \in \Omega \\ \sigma_{ij} = C_{ijkl}\epsilon_{kl} \quad & x \in \Omega \\ \epsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}) \quad & x \in \Omega \end{aligned} \tag{2.9}$$

where $\hat{f}_i = \rho f_i$. ρ describes the density of entire object. f is called a body force vector and defines forces that act on the entire body e.g. gravity. The first equation is called equation of motion.

Second is called constitutive equation. Third is called a geometric equation. All of them are linear PDE, which is very convenient. Linear PDE's are much easier to discretize using FEM. At the end of discretizing linear PDE we will obtain linear system of algebraic equations. However, the simplified linear formulation comes with a price. Large deformations cause visual deformation and should be usually be modelled using non linear equations.

$$\begin{aligned} u_i &= \hat{u_i} \quad x \in \partial\Omega_u \\ \sigma_{ij} n_j &= \hat{t_i} \quad x \in \partial\Omega_\sigma \end{aligned} \tag{2.10}$$

Dynamic System

In order to accommodate for the dynamic behaviour of the system, we must slightly change the equation of motion.

$$\sigma_{ij,j} + \hat{f_i} = \rho \ddot{u_i} \quad x \in \Omega \tag{2.11}$$

where ρ is a known quantity.

Weak formulation

Set of kinematically acceptable displacement fields or trial functions

$$\mathcal{P} = \{u(x) : u_i = \hat{u_i} \quad for \quad x \in \partial\Omega_u\} \tag{2.12}$$

Set of kinematically acceptable variations of function in $\mathcal{P}$.

$$\mathcal{W} = \{\delta u(x) : \delta u_i = 0 \quad for \quad x \in \partial\Omega_u\} \tag{2.13}$$

δu is called a virtual displacement.

For any variation $\delta u \in \mathcal{W}$ it is true that:

$$\int_\Omega (\sigma_{ij,j} + \hat{f_i}) \delta u_i dV - \int_{\partial\Omega_\sigma} (\sigma_{ij} n_j - \hat{t_i}) \delta u_i dA = 0 \tag{2.14}$$

Let us transform the second integral from Eq. 2.14, having in mind that $\partial\Omega = \partial\Omega_u \cup \partial\Omega_\sigma$. From definition of Eq. 2.13, we see that variation δu_i vanishes on remaining part of the boundary space $\partial\Omega_u$. Thus, we can treat this integral as a integral over entire boundary space $\partial\Omega$. Now we can apply Gauss-Ostrogradsky theorem and transform it to:

$$\int_{\partial\Omega_\sigma} (\sigma_{ij} n_j - \hat{t_i}) \delta u_i dA = \int_\Omega (\sigma_{ij,j} \delta u_i + \sigma_{ij} \delta u_{i,j}) dV - \int_{\partial\Omega_\sigma} \hat{t} \delta u_i dA \tag{2.15}$$

Substituting the above equation in Eq. 2.14 and naming $\delta\epsilon = \text{sym}\ \delta u_{i,j}$ we get:

$$\int_\Omega (\sigma_{ij} \delta\epsilon_{ij} dV) = \int_\Omega \hat{f_i} \delta u_i dV + \int_{\partial\Omega_\sigma} \hat{t_i} \delta u_i dA \tag{2.16}$$

The Eq. 2.16 is called the principle of virtual work for linear, static solid deformations. It claims that the work made by external forces on the virtual displacements(right hand side) is equal to the work made by interior forces(stress) on certain virtual displacement.

By including the constitutive equation and the symmetry of stiffness tensor, we can further transform the Eq. 2.16:

$$\int_\Omega (C_{ijkl} u_{k,l} \delta u_{i,j} dV) = \int_\Omega \hat{f}_i \delta u_i dV + \int_{\partial\Omega_\sigma} \hat{t}_i \delta u_i dA \tag{2.17}$$

Dynamic System

Once again, we want to add dynamic behaviour to the system. We do this by following that same steps but this time using the dynamic equation of motion(E.q 2.11).

$$\int_\Omega (C_{ijkl} u_{k,l} \delta u_{i,j} dV) + \int_\Omega \rho \ddot{u}_i \delta u_i dV = \int_\Omega \hat{f}_i \delta u_i dV + \int_{\partial\Omega_\sigma} \hat{t}_i \delta u_i dA \tag{2.18}$$

Discrete formulation

The displacement fields $u_i(x)$ and their variations $\delta u_i(x)$ must be properly included in classes $\mathcal{P}$ and $\mathcal{W}$. By the definitions of classes $\mathcal{P}$ and $\mathcal{W}$, the following must be true:

$$\forall_{u_i \in \mathcal{P}, \delta u_i \in \mathcal{W}} \quad u_i + \delta u_i \in \mathcal{P} \tag{2.19}$$

Further, we must include the boundary conditions:

$$\begin{aligned} u_i &= \hat{u}_i \quad x \in \partial\Omega_u \\ \delta u_i &= 0 \quad x \in \partial\Omega_u \end{aligned} \tag{2.20}$$

The displacement fields must also be once-differentiable as can be seen on the left hand side of Eq. 2.17.

Let us consider the following classes of displacement fields and their variations.

$$\begin{aligned} \mathcal{P}^{\mathcal{N}} &= \{\bar{u}(x) = \hat{\Phi}_i(x) + \Phi_{i\alpha}(x) q_\alpha\} \\ \mathcal{W}^{\mathcal{N}} &= \{\bar{\delta u}(x) = \Phi_{i\alpha}(x) \delta q_\alpha\} \end{aligned} \tag{2.21}$$

This is a linear combination of three systems of N linearly independent shape functions $\Phi_{i\alpha}(x)$, $i = 1, 2, 3$, with real coefficients q_α and δq_α. These shape functions satisfy the differentiable criteria. We choose such functions $\Phi_{i\alpha}(x)$ so that they satisfy the condition $\Phi_{i\alpha} = 0$ on $\partial\Omega_u$. Furthermore, $\hat{\Phi}_i(x)$ $i = 1, 2, 3$ are any functions satisfying the condition $\hat{\Phi}_i(x) = \hat{u}_i$ on $\partial\Omega_u$.

Let us substitute such displacements fields and their variations into Eq. 2.17.

$$\delta q_\alpha [q_\beta \int_\Omega C_{ijkl} \Phi_{i\alpha,j} \Phi_{k\beta,l} dV + \int_\Omega C_{ijkl} \Phi_{i\alpha,j} \hat{\Phi}_{k,l} dV - \int_\Omega \hat{f}_i \Phi_{i\alpha} dV - \int_{\partial\Omega_\sigma} \hat{t}_i \Phi_{i\alpha} dA] = 0 \tag{2.22}$$

Introducing new notation:

$$\begin{aligned} K_{\alpha\beta} &= \int_\Omega C_{ijkl}\Phi_{i\alpha,j}\Phi_{k\beta,l}dV \\ Q_\alpha &= -\int_\Omega C_{ijkl}\Phi_{i\alpha,j}\hat{\Phi}_{k,l}dV + \int_\Omega \hat{f}_i\Phi_{i\alpha}dV + \int_{\partial\Omega_\sigma}\hat{t}_i\Phi_{i\alpha}dA \end{aligned} \tag{2.23}$$

We can now rewrite Eq. 2.22:

$$\delta q_\alpha(K_{\alpha\beta}q_\beta - Q_\alpha) = 0 \tag{2.24}$$

The above equation must be true for all variation $\delta\bar{u}_i$, thus they are also true for all coefficients δq_α. Finally, the following system of equations must be true for all q_α coefficients:

$$K_{\alpha\beta}q_\beta = Q_\alpha \tag{2.25}$$

We can write this in matrix form:

$$K_{N\times N}q_{N\times 1} = Q_{N\times 1} \tag{2.26}$$

In Eq. 2.32 we are presented with a linear system of algebraic equations. Matrix K is called a stiffness matrix. Vector Q is called a exterior force vector, associated with displacements q. For now it will be only mentioned briefly that matrix K is a symmetric and sparse matrix. Details will follow.

Dynamic System

Let us consider more general case in which we include dynamic behaviour. The displacement fields are not functions of time.

$$\bar{u}(x,t) = \hat{\Phi}_i(x,t) + \Phi_{i\alpha}(x)q_\alpha(t) \tag{2.27}$$

The first and second derivative over time:

$$\begin{aligned} \dot{\bar{u}}(x,t) &= \dot{\hat{\Phi}}_i(x,t) + \Phi_{i\alpha}(x)\dot{q_\alpha}(t) \\ \ddot{\bar{u}}(x,t) &= \ddot{\hat{\Phi}}_i(x,t) + \Phi_{i\alpha}(x)\ddot{q_\alpha}(t) \end{aligned} \tag{2.28}$$

We can now state the discrete formulation for dynamic system using the Eq. 2.18.

$$\begin{aligned} &\delta q_\alpha[q_\beta\int_\Omega C_{ijkl}\Phi_{i\alpha,j}\Phi_{k\beta,l}dV \\ &+\int_\Omega C_{ijkl}\Phi_{i\alpha,j}\hat{\Phi}_{k,l}dV + \ddot{q_\beta}\int_\Omega \rho\Phi_{i\alpha}\Phi_{i\beta}dV + \int_\Omega \rho\Phi_{i\alpha}\ddot{\hat{\Phi}}_i dV - \int_\Omega \hat{f}_i\Phi_{i\alpha}dV - \int_{\partial\Omega_\sigma}\hat{t}_i\Phi_{i\alpha}dA] = 0 \end{aligned} \tag{2.29}$$

As before, Eq. 2.29 must be satisfied for all coefficients δq_α which means that everything else must vanish for all indices α. Let us denote mass matrix by:

$$M_{\alpha\beta} = \int_\Omega \rho \Phi_{i\alpha} \Phi_{j\beta} dV \tag{2.30}$$

The stiffness matrix K does not change, however we must change force vector Q.

$$Q_\alpha = -\int_\Omega C_{ijkl} \Phi_{i\alpha,j} \hat{\Phi}_{k,l} dV + \int_\Omega (\hat{f}_i - \rho \ddot{\hat{\Phi}}_i) \Phi_{i\alpha} dV + \int_{\partial\Omega_\sigma} \hat{t}_i \Phi_{i\alpha} dA \tag{2.31}$$

We can expand our system of linear equations to:

$$M_{N\times N} \ddot{q}_{N\times 1} + K_{N\times N} q_{N\times 1} = Q_{N\times 1} \tag{2.32}$$

This is a linear system of differential equations of second order. The unknowns being the displacement fields $q_\alpha(t)$, which must be satisfied for each time t. Details on solving this system will follow.

The final part of dynamic system is the damping matrix C. Damping is introduced to simulate friction forces internal to the structure. One model of creating damping matrix C for structural systems is called Rayleigh damping matrix [**?**]:

$$C = \alpha M + \beta K \tag{2.33}$$

In other words, C is the linear combination of mass and stiffness matrices.

Thus the final form of dynamic system is presented here:

$$M_{N\times N} \ddot{q}_{N\times 1} + C_{N\times N} \dot{q}_{N\times 1} + K_{N\times N} q_{N\times 1} = Q_{N\times 1} \tag{2.34}$$

Finite Element Method

We will divide the space Ω using E finite elements, each represented by the set Ω_e, $e = 1, 2, ..., E$, where $\Omega_e \cap \Omega_f = \emptyset$, for $e \neq f$.

Boundary of e-th element will be denoted $\partial\Omega_e$. Common boundary of neighbours e and f will be denoted $\partial\Omega_{ef} = \partial\Omega_e \cap \partial\Omega_f$. The part of elements boundary that also happens to be boundary of the entire body will be denoted $\partial\Omega_{\bar{e}} = \partial\Omega \cap \partial\Omega_e$.

We can use the finite elements to compute the integral over the entire body.

$$\int_\Omega (.) dV = \sum_{e=1}^{E} \int_{\Omega_e} (.) dV \tag{2.35}$$

Similarly for the integrals over the area:

$$\int_{\partial\Omega} (.) dA = \sum_{\bar{e}\in\{E_{\partial\Omega}\}} \int_{\partial\Omega_{\bar{e}}} (.) dA \tag{2.36}$$

where $E_{\partial\Omega}$ is a set of finite elements which have non empty $\partial\Omega_{\bar{e}}$ set.

In FEM models we can apply physical intuition behind the discrete formulation. Recall that the shape functions $\Phi_\alpha(x)$ were associated with certain real coefficients q_α. These coefficients will now be associated with vertices x_α of the finite elements. The coefficient q_α will now be called parameter of vertex x_α. In 3D solid mechanics the parameter q_α denotes the component of displacements vector of the associated vertex x_α. Formally the displacement field will be approximated by:

$$\bar{u}(x) = \Phi_\alpha(x) q_\alpha \tag{2.37}$$

for $\alpha = 1, 2, ..., N$, where N is the number of vertices $x_\alpha \in \Omega$.

Moreover, we can require that the coefficients q_α will be the value of approximated displacement fields at their associated vertices x_α:

$$\bar{u}(x_\alpha) = q_\alpha \tag{2.38}$$

We can easily design a shape function satisfying the above criteria:

$$\Phi_\alpha(x_\beta) = \delta_{\alpha\beta} \tag{2.39}$$

for $\alpha, \beta = 1, 2, ..., N$. In other words, the shape function $\Phi_\alpha(x)$ associated with vertex x_α should have value $\Phi_\alpha(x_\alpha) = 1$ and value 0 in any other vertex. Moreover, we usually require the shape function to sum up to 1:

$$\sum_{\alpha=1}^{N} \Phi_\alpha(x) = 1 \tag{2.40}$$

FEM for Solid Mechanics

For 3D solid mechanics, the approximated displacement field has a form of:

$$\bar{u}_i(x) = \Phi_{i\alpha}(x) q_\alpha \tag{2.41}$$

where $i = 1, 2, 3$ and q_α denotes the values of displacements in associated vertices. However, since in 3D the displacement vector is described by 3 components(x, y, z), we must expand the range of α index. For model with N vertices, $\alpha = 1, 2, ...3N$. We can now represent the Eq. 2.42 in matrix form:

$$\bar{u}_{3\times 1}(x) = \Phi^T_{3\times 3N}(x) q_{3N\times 1} \tag{2.42}$$

That makes our system of linear equations:

$$K_{3N\times 3N} q_{3N\times 1} = Q_{3N\times 1} \tag{2.43}$$

where

$$\begin{aligned} K_{\alpha\beta} &= \int_\Omega C_{ijkl}\Phi_{i\alpha,j}\Phi_{k\beta,l}dV \\ Q_\alpha &= \int_\Omega \hat{f}_i\Phi_{i\alpha}dV + \int_{\partial\Omega_\sigma} \hat{t}_i\Phi_{i\alpha}dA \end{aligned} \tag{2.44}$$

We can use the fact that C_{ijkl} is a symmetric tensor to further simplify the equations. We will represent second order tensor with 6×1 vector and a fourth order tensor with 6×6 matrix.

$$\begin{aligned} K_{3N\times 3N} &= \int_\Omega B^T_{6\times 3N} C_{6\times 6} B_{6\times 3N} dV \\ Q_{3N\times 1} &= \int_\Omega \Phi_{3N\times 3}\hat{f}_{3\times 1}dV + \int_{\partial\Omega_\sigma} \Phi_{3N\times 3}\hat{t}_{3\times 1}dA \end{aligned} \tag{2.45}$$

Where $B_{6\times 3N}$ is called a geometric matrix and is defined as follows:

$$\begin{aligned} [B_{6\times 3N}] &= [B^{(1)}_{6\times 3} B^{(2)}_{6\times 3} ... B^{(N)}_{6\times 3}] \\ [B^{(\bar{\alpha})}_{6\times 3}] &= \begin{bmatrix} \Phi^{\bar{\alpha}}_{,1} & & \\ & \Phi^{\bar{\alpha}}_{,2} & \\ & & \Phi^{\bar{\alpha}}_{,3} \\ \Phi^{\bar{\alpha}}_{,2} & \Phi^{\bar{\alpha}}_{,1} & \\ & \Phi^{\bar{\alpha}}_{,3} & \Phi^{\bar{\alpha}}_{,2} \\ \Phi^{\bar{\alpha}}_{,3} & & \Phi^{\bar{\alpha}}_{,1} \end{bmatrix} \end{aligned} \tag{2.46}$$

If in a given finite element e there are N_e vertices then the local count of vertex parameters is equal to $3N_e$. We will now introduce local stiffness matrix $k^{(e)}_{3N_e\times 3N_e}$ and force vector $p^{(3)}_{3N_e\times 1}$. These are computed using the Eq. 2.45 but using only the local shape functions of each finite element, namely the non-zero values on Ω_e section. However, we must somehow map these local values to our global values, K and Q. We introduce partial global stiffness matrix $K^{(e)}_{3N\times 3N}$ and partial global force vector $Q^{(e)}_{3N\times 1}$. They contain the local values(k^e and $p^{(e)}$) of the finite element e, mapped into their global coordinates using boolean matrix $A^{(e)}_{3N_e\times 3N}$.

$$\begin{aligned} K^{(e)}_{\alpha\beta} &= A^{(e)}_{a\alpha} k^{(e)}_{ab} A^{(e)}_{b\beta} \\ Q^{(e)}_\alpha &= p^{(e)}_a A^{(e)}_{a\alpha} \end{aligned} \tag{2.47}$$

These partial global matrices are then summed into global matrices K and Q.

$$\begin{aligned} K_{\alpha\beta} &= \sum_{e=1}^{E} K^{(e)}_{\alpha\beta} \\ Q_\alpha &= \sum_{e=1}^{E} Q^{(e)}_\alpha \end{aligned} \tag{2.48}$$

Dynamic System

For dynamic system we must also include mass matrix M introduced in Eq. 2.30. To complete the matrix form of computing K and Q in Eq. 2.45 we now provide the matrix equation for M:

$$M_{3N\times 3N} = \int_{\Omega} \rho \Phi^T_{3N\times 3} \Phi_{3N\times 3} dV \tag{2.49}$$

We compute the local mass matrix m_e using Eq. 2.49 over the Ω_e section. Assembly process is exactly the as in the case of stiffness matrix.

Mass Matrix Diagonalization

TODO

Sparse Matrix Representation

There exists many methods for storing sparse matrices in efficient way. One of the most popular methods is called Compressed Sparse Row(CSR). This method is chosen because of its common usage in numerical libraries in CUDA API. We will learn about its programming usages later on. For now we provide the definition for CSR storage.

For simplicity we assume that a matrix M is square sparse $n \times n$ matrix. CSR format stores matrix M using three one-dimensional vectors A_V, A_I, A_J. Denote the number of non-zero elements in M by n_z. The vectors are constructed as follows:

1. The vector A_V is of length n_z and holds all the non-zero elements of M in left-to-right top-to-bottom(row-major) order.
2. The vector A_J is of length n_z. Contains the column indices in M of each element in A_V.
3. The vector A_I is of length $n + 1$ and is defined as follows:
   (a) $A_I[0] = 0$
   (b) $A_I[i] = A_I[i-1]+$ (the number of non-zero elements on the $i-1$th row in M)

Example 2.1. Let M be a sparse matrix:

$$M = \begin{bmatrix} 1 & -1 & 0 & -3 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{bmatrix} \tag{2.50}$$

Matrix M in CSR format looks like the following:

Let M be a matrix:

$$\begin{aligned} A_V &= \begin{bmatrix} 1 & -1 & -3 & -2 & 5 & 4 & 6 & 4 & -4 & 2 & 7 & 8 & -5 \end{bmatrix} \\ A_J &= \begin{bmatrix} 0 & 1 & 3 & 0 & 1 & 2 & 3 & 4 & 0 & 2 & 3 & 1 & 4 \end{bmatrix} \\ A_I &= \begin{bmatrix} 0 & 3 & 5 & 8 & 11 & 13 \end{bmatrix} \end{aligned} \tag{2.51}$$

We can further improve the efficiency if we know that matrix M is also symmetric. In such a case we only store the upper or the lower triangle of matrix M. One important restrictions rises, namely the fact that all diagonal values must be stored in CSR format, including zero elements.

Example 2.2. Let M be a sparse and symmetric matrix:

$$M = \begin{bmatrix} 1 & -1 & 0 & -3 & 0 \\ -1 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -3 & 0 & 6 & 7 & 0 \\ 0 & 0 & 4 & 0 & -5 \end{bmatrix} \tag{2.52}$$

Matrix M in Symmetric-CSR format looks like the following:

Let M be a matrix:

$$\begin{aligned} A_V &= \begin{bmatrix} 1 & -1 & -3 & 5 & 4 & 6 & 4 & 7 & -5 \end{bmatrix} \\ A_J &= \begin{bmatrix} 0 & 1 & 3 & 1 & 2 & 3 & 4 & 3 & 4 \end{bmatrix} \\ A_I &= \begin{bmatrix} 0 & 3 & 4 & 7 & 8 & 9 \end{bmatrix} \end{aligned} \tag{2.53}$$

Solver

Implicit Euler

Let us rewrite the Eq. 2.34:

$$Ma + Cv + K(x - \bar{x}) = Q \tag{2.54}$$

where a is the acceleration, v is the velocity, x is the current positions and $\bar{x}$ is the initial position. Notice that the displacement q is defined as $x - \bar{x}$. We will use the implicit Euler scheme presented in Eq. 1.17 to integrate this system. Formally we want to compute:

$$\begin{aligned} v^+ &= v + \Delta t a^+ \\ x^+ &= x + \Delta t v^+ \end{aligned} \tag{2.55}$$

where x^+, v^+, a^+ denote respectively new positions, new velocities and new accelerations. Substituting into Eq. 2.54 and rearranging yields:

$$(M + \Delta tC + \Delta t^2 K)v^+ = \Delta tQ + Mv - \Delta tK(x - \bar{x}) \tag{2.56}$$

Which we can write shorter:

$$Av^+ = b \tag{2.57}$$

The above equation is solved for v^+ and is used to compute x^+. Recall that matrices M, C and K are constant over all time steps. Thus, the left hand side A can be precomputed and used between all time steps. It is also important to state that left hand side A remains its structure. Namely it is still symmetric and sparse matrix. To compute right hand side b we have to do 3 operations every time step. First, trivial scalar vector multiplication ΔtQ. Second, a diagonal matrix and vector multiplication Mv. Finally, the most computation intensive out of the three, symmetric sparse matrix and vector multiplication $\Delta tK(x - \bar{x})$.

Boundary Conditions

In both static and dynamic systems we end up with the system of linear equations of the form: $Ax = b$. In the case of static system, x is a vector of displacements. However in the implicit dynamic solver x is a vector of the first derivative of displacements, i.e. velocities. In any case, applying the Dirichlet boundary conditions are done exactly the same, however they have a different meaning. In the former case we apply conditions directly on the values of displacements. In the latter, we apply conditions on their velocities. Method of applying the Dirichlet boundary conditions are presented.

Algorithm 3 Dirichlet BC for $Ax = b$

1: **procedure** Dirichlet BC
2: **for** each bc $\bar{x}_j$ **do**
3: Subtract from each ith member of b, the product of A_{ij} and x_j: $\hat{b_i} = b_i - A_{ij}\bar{x_j}$.
4: Zero the jth row and column of A: $\hat{A}_{ij} = \hat{A}_{ji} = 0$
5: Set $\hat{A_{jj}} = 1$
6: Set $b_j = \bar{x}_j$

Example 2.3. Consider the following system.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.58}$$

After applying boundary conditions with a known value $\bar{x_3}$, the system becomes:

$$\begin{bmatrix} A_{11} & A_{12} & 0 \\ A_{21} & A_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 - A_{13}\bar{x_3} \\ b_2 - A_{23}\bar{x_3} \\ \bar{x_3} \end{bmatrix} \tag{2.59}$$

Solid Mechanics - Part 2 - Tetrahedron

One of the most simple finite elements used in FEM models is a linear tetrahedron. Such element consists of four nodes and its shape functions are linear polynomials. This finite element is in particular interested for real time applications since no numerical integration are needed to construct element equations.

The tetrahedron is defined by four vertices with components x_i, y_i, z_i coordinates, $i = 1, 2, 3, 4$, six edges and four faces. For simplicity we can denote the component differences: $x_{ij} = x_i - x_j$, $y_{ij} = y_i - y_j$, $z_{ij} = z_i - z_j$ for $i, j = 1, 2, 3, 4$. The vertices can not be coplanar.

Volume

We can use the Jacobian matrix J to compute the volume V of the tetrahedron.

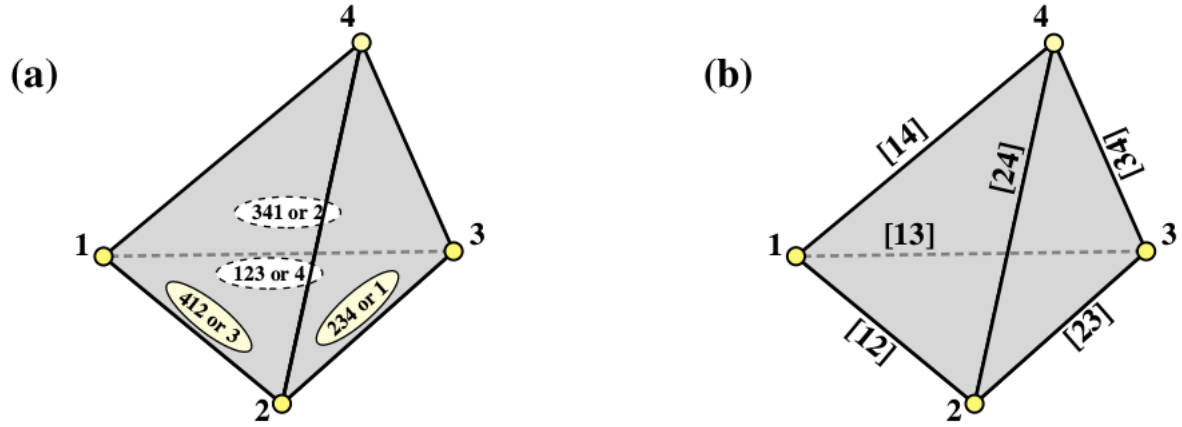
$$V = \int_{\Omega_e} d\Omega_e = \frac{1}{6} det \begin{bmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix} = \frac{1}{6} det(J) \tag{3.1}$$

The vertices are coplanar when V is equal to zero. The vertices have indices: $1, 2, 3, 4$. Edges are denoted by pair of indices e.g. 23 is an edge from vertex 2 to vertex 3. Faces are denoted by their opposite vertex or by triple of vertex indices that make up this face e.g. 1 or 234.

Natural Coordinates

So far we have specified the tetrahedron vertices in Cartesian coordinates x, y, z. An alternative coordinate system is called tetrahedral natural coordinates and is composed of four functions: $\Phi^{(1)}(x), \Phi^{(2)}(x), \Phi^{(3)}(x), \Phi^{(4)}(x)$. They value of $\Phi^{(i)}$ is equal to i at vertex 1 and 0 in all other vertices. We add a constraint:

$$\Phi^{(1)} + \Phi^{(2)} + \Phi^{(3)} + \Phi^{(4)} = 1 \tag{3.2}$$



Rysunek 3.1: Naming conventions for tetrahedron faces (a) and edges (b). Courtesy of [5]

Transformation

We have defined a different coordinate system. However, all quantities such as displacement fields, strain or stress are expressed in Cartesian coordinate system. Thus we need construct a transformation between this two coordinate systems. We combine the identity constraint in Eq 3.2 with the linear interpolation of natural coordinates i.e. $x = x_i\Phi^{(i)}$, $y = y_i\Phi^{(i)}$, $z = z_i\Phi^{(i)}$ to get the following matrix relation:

$$\begin{bmatrix} 1 \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix} = \begin{bmatrix} \Phi^{(1)} \\ \Phi^{(2)} \\ \Phi^{(3)} \\ \Phi^{(4)} \end{bmatrix} \tag{3.3}$$

The inversion of the above system yields:

$$\begin{bmatrix} \Phi^{(1)} \\ \Phi^{(2)} \\ \Phi^{(3)} \\ \Phi^{(4)} \end{bmatrix} = \frac{1}{6V} \begin{bmatrix} 6V_{01} & y_{42}z_{32} - y_{32}z_{42} & x_{32}z_{42} - x_{42}z_{32} & x_{42}y_{32} - x_{32}y_{42} \\ 6V_{02} & y_{31}z_{43} - y_{34}z_{13} & x_{43}z_{31} - x_{13}z_{34} & x_{31}y_{43} - x_{34}y_{13} \\ 6V_{03} & y_{24}z_{14} - y_{14}z_{24} & x_{14}z_{24} - x_{24}z_{14} & x_{24}y_{14} - x_{14}y_{24} \\ 6V_{04} & y_{13}z_{21} - y_{12}z_{31} & x_{21}z_{13} - x_{31}z_{12} & x_{13}y_{21} - x_{12}y_{31} \end{bmatrix} \begin{bmatrix} 1 \\ x \\ y \\ z \end{bmatrix} \tag{3.4}$$

The first column is abbreviation for:

$$\begin{aligned} 6V_{01} &= x_2(y_3z_4 - y_4z_3) + x_3(y_4z_2 - y_2z_4) + x_4(y_2z_3 - y_3z_2) \\ 6V_{02} &= x_1(y_4z_3 - y_3z_4) + x_3(y_1z_4 - y_4z_1) + x_4(y_3z_1 - y_1z_3) \\ 6V_{03} &= x_1(y_2z_4 - y_4z_2) + x_2(y_4z_1 - y_1z_4) + x_4(y_1z_2 - y_2z_1) \\ 6V_{04} &= x_1(y_3z_2 - y_2z_3) + x_2(y_1z_3 - y_3z_1) + x_3(y_2z_1 - y_1z_2) \end{aligned} \tag{3.5}$$

It can be shown that $V = V_{01} + V_{02} + V_{03} + V_{04}$.

We can now write the matrix system in more compact way using further abbreviation for the other part of the 4×4 matrix:

$$\begin{bmatrix} \Phi^{(1)} \\ \Phi^{(2)} \\ \Phi^{(3)} \\ \Phi^{(4)} \end{bmatrix} = \frac{1}{6V} \begin{bmatrix} 6V_{01} & a_1 & b_1 & c_1 \\ 6V_{02} & a_2 & b_2 & c_2 \\ 6V_{03} & a_3 & b_3 & c_3 \\ 6V_{04} & a_4 & b_4 & c_4 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ y \\ z \end{bmatrix} \tag{3.6}$$

The explicit equation for $\Phi^{(i)}$ is given by:

$$\Phi^{(i)} = \frac{6V_{0i} + a_i x + b_i y + c_i z}{6V} \tag{3.7}$$

Derivatives

We can easily compute the following partial derivatives

$$\begin{aligned} 6V \frac{\partial \Phi^{(i)}}{\partial x} &= 6V \Phi^{(i)}_{,1} = a_i \\ 6V \frac{\partial \Phi^{(i)}}{\partial y} &= 6V \Phi^{(i)}_{,2} = b_i \\ 6V \frac{\partial \Phi^{(i)}}{\partial z} &= 6V \Phi^{(i)}_{,3} = c_i \end{aligned} \tag{3.8}$$

Analytical Integration

As mentioned before, integration over the linear tetrahedron can be done analytically using the general formula:

$$\int_{\Omega_e} \Phi^{i(1)} \Phi^{j(2)} \Phi^{k(3)} \Phi^{l(4)} d\Omega_e = \frac{i!j!k!l!}{(i+j+k+l+3)!} 6V \tag{3.9}$$

Here the indices without brackets i, j, k, l represent the power exponent. Special cases that will be of interest for us:

$$\begin{aligned} \int_{\Omega_e} d\Omega_e &= V \\ \int_{\Omega_e} \Phi^{(i)} d\Omega_e &= \frac{1}{4} V \\ \int_{\Omega_e} \Phi^{(i)} \Phi^{(j)} d\Omega_e &= \begin{cases} \frac{1}{10} V & i = j \\ \frac{1}{20} V & i \neq j \end{cases} \end{aligned} \tag{3.10}$$

We now have all components needed to compute local stiffness matrix $k_{12\times12}$ and local force vector $p_{12\times1}$. Our natural coordinates $\Phi^{(i)}$ will play a role of shape functions.

Stiffness Matrix

First, we will define geometric matrix B mentioned in Eq. 3.16 explicitely in terms of derivatives computed in Eq. 3.8:

$$B_{6\times12} = \frac{1}{6V}\begin{bmatrix} a_1 & 0 & 0 & a_2 & 0 & 0 & a_3 & 0 & 0 & a_4 & 0 & 0 \\ 0 & b_1 & 0 & 0 & b_2 & 0 & 0 & b_3 & 0 & 0 & b_4 & 0 \\ 0 & 0 & c_1 & 0 & 0 & c_2 & 0 & 0 & c_3 & 0 & 0 & c_4 \\ b_1 & a_1 & 0 & b_2 & a_2 & 0 & b_3 & a_3 & 0 & b_4 & a_4 & 0 \\ 0 & c_1 & b_1 & 0 & c_2 & b_2 & 0 & c_3 & b_3 & 0 & c_4 & b_4 \\ c_1 & 0 & a_1 & c_2 & 0 & a_2 & c_3 & 0 & a_3 & c_4 & 0 & a_4 \end{bmatrix} \tag{3.11}$$

We are going to use the Eq. 2.45 with integrals over Ω_e.

First let us compute $k^{(e)}$

$$k^{(e)}_{12\times12} = \int_{\Omega_e} B^T_{6\times12} C_{6\times6} B_{6\times12} dV \tag{3.12}$$

Since both B and C are constant i.e. do not depend on x we simply get:

$$k^{(e)} = VB^TCB \tag{3.13}$$

Force Vector

We split the force into two independent parts: body forces and traction forces.

$$p_{12\times1} = \int_{\Omega_e} \Phi^T_{3\times12} \hat{f}_{3\times1} dV + \int_{\partial\Omega_{\sigma_e}} \Phi^T_{3\times12} \hat{t}_{3\times1} dA \tag{3.14}$$

Body Forces

Body forces such as gravity, are defined are a single force vector $\hat{f} = \rho[f_1, f_2, f_3]$. This force is applied to all vertices of body and is weighted by the volume of the tetrahedron that the vertex belongs to. The body force is computed using the fist integral:

$$f^{(e)}_{12\times1} = \int_{\Omega_e} \Phi^T_{3\times12} \hat{f}_{3\times1} dV \tag{3.15}$$

Even if we assume that the body force $\hat{f}$ is constant, Φ is not constant. It depends on the shape functions $\Phi^{(i)}$ which in turn depends on x. Let us define Φ explicitly:

$$\Phi_{3\times 12} = \begin{bmatrix} \Phi^{(1)} & 0 & 0 & \Phi^{(2)} & 0 & 0 & \Phi^{(3)} & 0 & 0 & \Phi^{(4)} & 0 & 0 \\ 0 & \Phi^{(1)} & 0 & 0 & \Phi^{(2)} & 0 & 0 & \Phi^{(3)} & 0 & 0 & \Phi^{(4)} & 0 \\ 0 & 0 & \Phi^{(1)} & 0 & 0 & \Phi^{(2)} & 0 & 0 & \Phi^{(3)} & 0 & 0 & \Phi^{(4)} \end{bmatrix} \tag{3.16}$$

We can use the second analytical integral presented in Eq. 3.10. Assuming that the force $\hat{f} = \rho[f_1, f_2, f_3]$ is constant we receive:

$$f^{(e)}_{12\times 1} = \frac{1}{4}\rho V[f_1, f_2, f_3, f_1, f_2, f_3, f_1, f_2, f_3, f_1, f_2, f_3]^T \tag{3.17}$$

Traction Forces

Traction forces simulate the effect of pressure load, e.g. load applied after collision. As opposed to body forces, the traction forces are applied to element faces along their unit normal vector. Thus, each tetrahedron element should have an input of four scalar traction force magnitudes p_i, $i = 1, 2, 3, 4$, for each face. Traction force for entire tetrahedron is result of a sum of four traction forces computed for each face. To compute traction force for face 1 or 234 with input magnitude p, we use the formula:

$$t^{(e)}_{12\times 1} = \frac{1}{3}pA_1[0, 0, 0, \bar{a_1}, \bar{b_1}, \bar{c_1}, \bar{a_1}, \bar{b_1}, \bar{c_1}, \bar{a_1}, \bar{b_1}, \bar{c_1}]^T \tag{3.18}$$

First we see that the vertex 1, opposite of face 234, has received total force equal to $[0, 0, 0]$. Then, all the other vertices receive the load equally spread among them $\frac{1}{3}pA_1$, where A_1 is the area of face 234. Further, the entire vector is directed along the direction cosines $\bar{a_1} = \frac{a_1}{S_1}, \bar{b_1} = \frac{b_1}{S_1}, \bar{c_1} = \frac{c_1}{S_1}$, where $S_1 = \sqrt{a_1^2 + b_1^2 + c_1^2}$. To compute the area A_1 we can use the cross product property. We choose two directed vectors from face 234 coming from any of its corners e.g. $u_{32} = [x_{32}, y_{32}, z_{32}]$ and $u_{42} = [x_{42}, y_{42}, z_{42}]$. Then:

$$A_1 = \frac{1}{2}||u_{32} \times u_{42}||_2 \tag{3.19}$$

The process is generalized for all face indices $i = 1, 2, 3, 4$.

Mass Matrix

Recalling the equation 2.49 for mass matrix M, we can compute the local mass matrix $m^{(e)}$ following the same steps as in the case of stiffness matrix.

$$m^{(e)}_{12\times 12} = \int_{\Omega_e} \rho \Phi^T_{3\times 12}\Phi_{3\times 12} dV \tag{3.20}$$

Using the equation 3.10 we receive:

$$m^{(e)}_{12\times12} = \frac{\rho V}{20}\begin{bmatrix} 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 2 \end{bmatrix} \tag{3.21}$$

Similarly to stiffness, mass matrix is also symmetric.

Mass Matrix Diagonalization

TODO

Solving System of Linear Equations

Conjugate Gradient

This section explains all the tools required to define the Conjugate Gradient algorithm.

Introduction

Conjugate Gradient(CG) is one of the most popular iterative methods of solving system of linear equations:

$$Ax = b \tag{4.1}$$

where A is a square $n \times n$, symmetric, positive-definite(or positive-indefinite) matrix, x and b are vectors e.i. $n \times 1$ matrices We define inner product(also known as dot product) of vector by

$$x^T y = \sum_{i=1}^{n} x_i y_i \tag{4.2}$$

Two vector x and y are said to be orthogonal when:

$$x^T y = 0 \tag{4.3}$$

A matrix A is positive-definite if for every nonzero vector x:

$$x^T A x > 0 \tag{4.4}$$

Quadratic Form

A quadratic form a scalar function of a vector:

$$f(x) = \frac{1}{2} x^T A x - b^T x + c \tag{4.5}$$

where A is matrix, x and b are vector and c is a scalar. Gradient of quadratic form is defined:

$$f'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ . \\ . \\ . \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix} \tag{4.6}$$

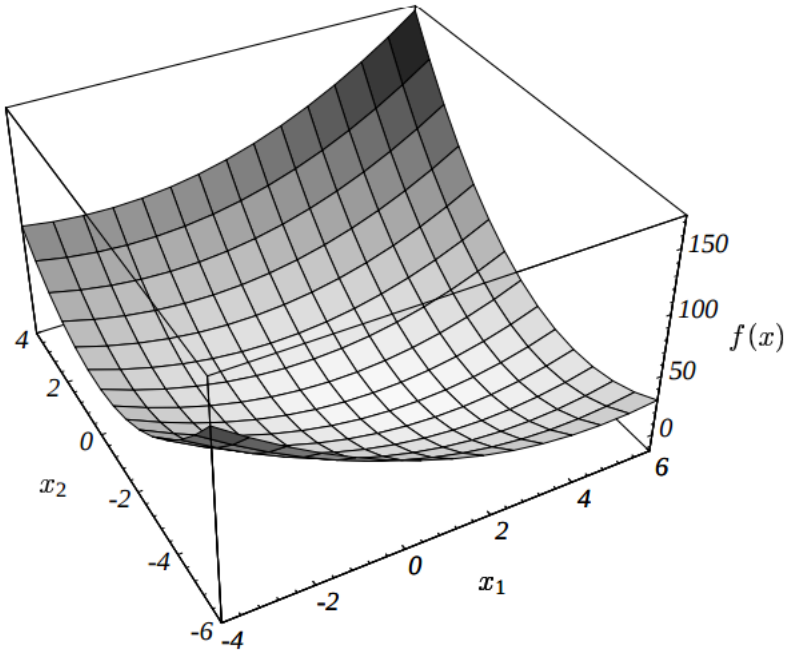
Taking a gradient of Eq. 4.5 yields:

$$f'(x) = \frac{1}{2} A^T x + \frac{1}{2} Ax - b \tag{4.7}$$

If A is symmetric then:

$$f'(x) = Ax - b \tag{4.8}$$

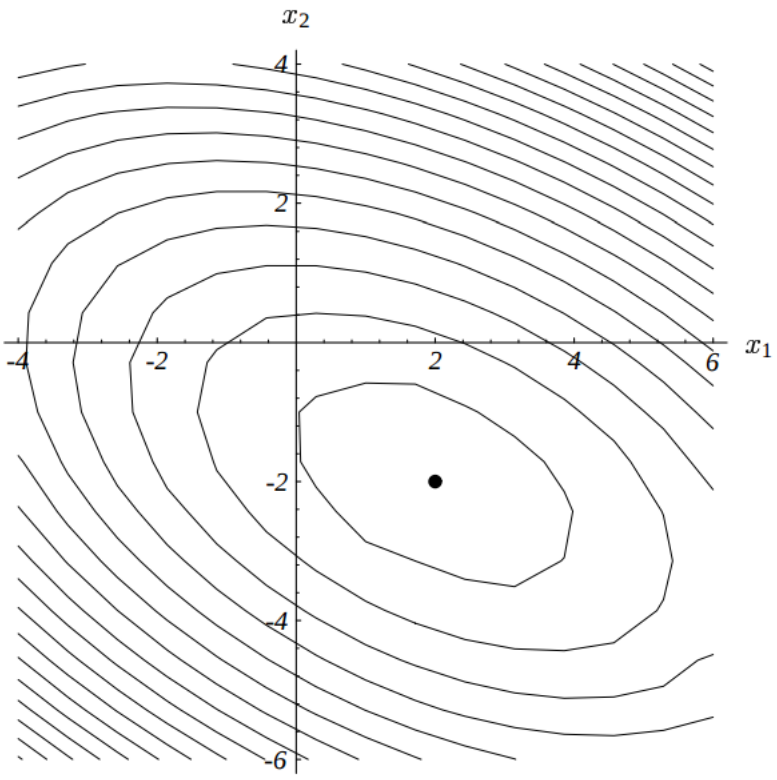
If we set the gradient to zero then we get the solution for $Ax = b$ which is also a critical point of $f(x)$. If A is also positive-definite then this solution is also a minimum of $f(x)$. In other words, to solve $Ax = b$, we can find such an x that minimizes $f(x)$.



Rysunek 4.1: Paraboloid of example quadratic form f(x). Courtesy of [7]

Steepest Descent

In the method of Steepest Descent we look for the solution x by going down the paraboloid. We start at some point x_0 and iterate the steps taking new points $x_1, x_2, ...$ until we hit the



Rysunek 4.2: Contour plot of example quadratic form f(x). Courtesy of [7]

minimum. Obviously we want to take the steps in the direction for which values of f decreases the most. That direction is the opposite of $f'(x)$, namely:

$$-f'(x) = b - Ax \tag{4.9}$$

Let us define a few measurements. The error:

$$e_i = x_i - x \tag{4.10}$$

Error vector e_i shows the distance, in a given iteration i, from the solution x.

The residual:

$$r_i = b - Ax_i \tag{4.11}$$

Residual vector r_i shows the distance, in a given iteration i, from correct value b. Combining equations 4.11 and 4.9 we get:

$$r_i = -f'(x_i) \tag{4.12}$$

That means that the residual is the direction of steepest descent. Thus the next step x_i is defined as:

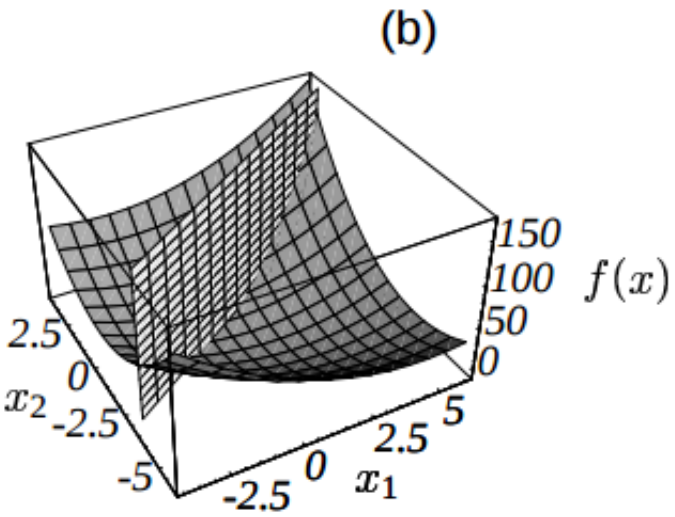
$$x_{i+1} = x_i + \alpha r_i \tag{4.13}$$

New issue arises, namely computing the length of direction vector r_i: α. We use a line search that will choose α that minimizes f along a line. Figures 4.3 and 4.3 show an example case of finding alpha. First in figure 4.3 we see the intersection of a paraboloid and a surface defined by some initial point x_0 and the direction of steepest descent. Figure 4.4 shows the parabola of this intersection, which helps us to see what is the value of α at the base of the parabola. This is obviously the case when the derivative $\frac{d}{d\alpha}f(x_{i+1})$ is equal to zero. We can use the chain rule to further compute:

$$\frac{d}{d\alpha}f(x_{i+1}) = f'(x_{i+1})^T \frac{d}{d\alpha}x_{i+1} = f'(x_{i+1})^T r_i \tag{4.14}$$

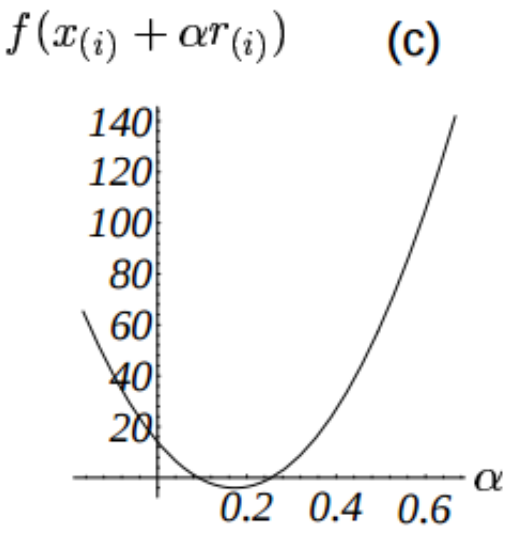
Setting the expression $f'(x_{1i+1})^T r_i$ to zero means that $f'(x_{i+1})^T$ and r_i have to orthogonal. Formally, computing α goes as follows:

$$\begin{aligned} r_{i+1}^T r_i &= 0 \\ (b - Ax_{i+1})^T r_i &= 0 \\ (b - A(x_i + \alpha r_i))^T r_i &= 0 \\ (b - Ax_i)^T r_i - \alpha(Ar_i)^T r_i &= 0 \\ (b - Ax_i)^T r_i &= \alpha(Ar_i)^T r_i \\ r_i^T r_i &= \alpha r_i^T (Ar_i) \\ \alpha &= \frac{r_i^T r_i}{r_i^T (Ar_i)} \end{aligned} \tag{4.15}$$



Rysunek 4.3: Paraboloid and a surface defined by residual direction at some step. Courtesy of [7]

The entire Steepest Descent algorithm can be summarized as follows:



Rysunek 4.4: The intersection parabola of two surfaces shown in 4.3. Courtesy of [7]

$$\begin{aligned} r_i &= b - Ax_i \\ \alpha_i &= \frac{r_i^T r_i}{r_i^T (Ar_i)} \\ x_{i+1} &= x_i + \alpha_i r_i \end{aligned} \tag{4.16}$$

Example solution of Steepest Descent is shown in figure 4.5

Conjugate Directions

Iterations in Steepest Descent method usually end up taking the same directions as one of the previous iterations, see figure 4.5. Conjugate Direction method chooses a set of orthogonal search directions $d_0, d_1, ..., d_{n-1}$. Taking next step looks now as follows:

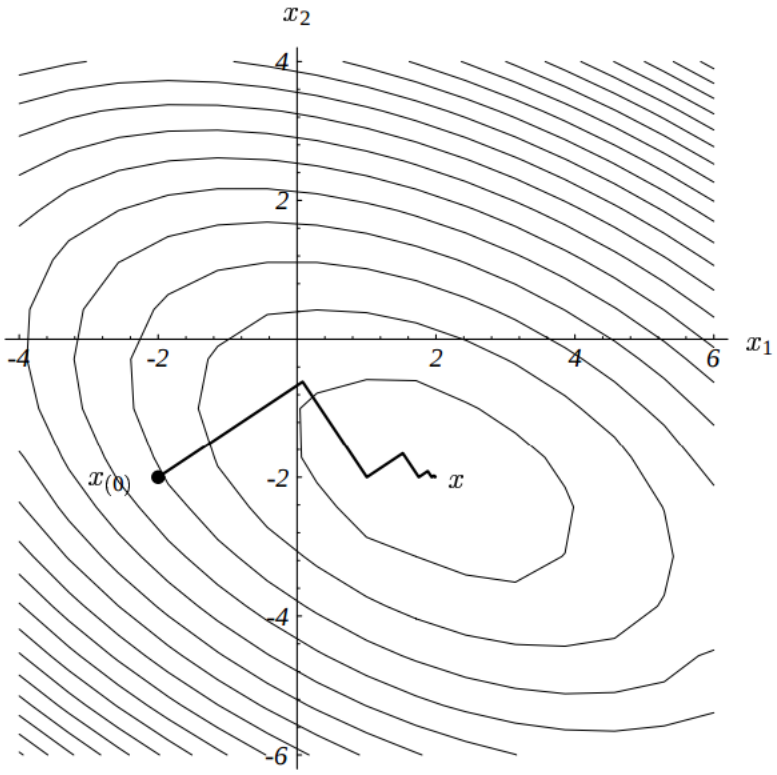
$$x_{i+1} = x_i + \alpha_i d_i \tag{4.17}$$

Generating the set of directions $\{d_i\}$ is done using the process called conjugate Gram-Schmidt. Assume we have n linearly independent vectors $u_0, u_1, ..., u_{n-1}$. We construct d_i using the formula:

$$d_i = u_i + \sum_{k=0}^{i-1} \beta_{ik} d_k \tag{4.18}$$

where β_{ij} is defined by:

$$\beta_{ij} = -\frac{u_i^T A d_j}{d_j^T A d_j} \tag{4.19}$$



Rysunek 4.5: Example of Steepest Descent solution. Courtesy of [7]

Conjugate Gradient

Finally we can define the Conjugate Gradient method. CG is a special case of Conjugate Directions method using $u_i = r_i$. The equations required to compute CG are presented:

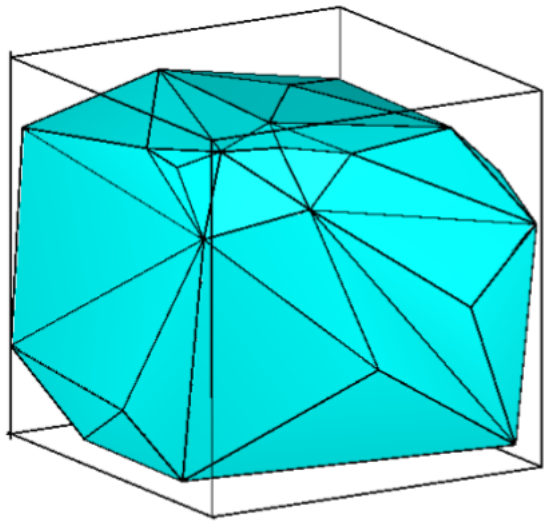
$$\begin{aligned} d_0 = r_0 &= b - Ax_0 \\ \alpha_i &= \frac{r_i^T r_i}{d^T i A d_i} \\ x_{i+1} &= x_i + \alpha_i d_i \\ r_{i+1} &= r_i - \alpha_i A d_i \\ \beta_{i+1,i} &= -\frac{r_{i+1}^T r_{i+1}}{r_i^T r_i} \\ d_{i+1} &= r_{i+1} + \beta_{i+1,i} d_i \end{aligned} \tag{4.20}$$

The most expensive operation in CG is the matrix-vector multiplication which in general has time complexity of $\mathcal{O}(m)$, where m is the number of non-zero elements in A. This means that each iterations of CG method has time complexity $\mathcal{O}(m)$.

The algorithm ends when maximum number of iteration has been reached or when r_i is small enough.

Tetrahedralization

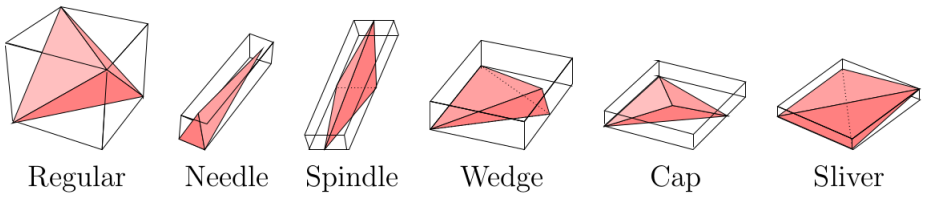
A quality mesh can be defined by a few properties. One of the most basic properties is the fact that generated 3D mesh should correctly model the shape of the input domain. Figure 5.1 shows an tetrahedron mesh that does not model the shape of the cube domain correctly.



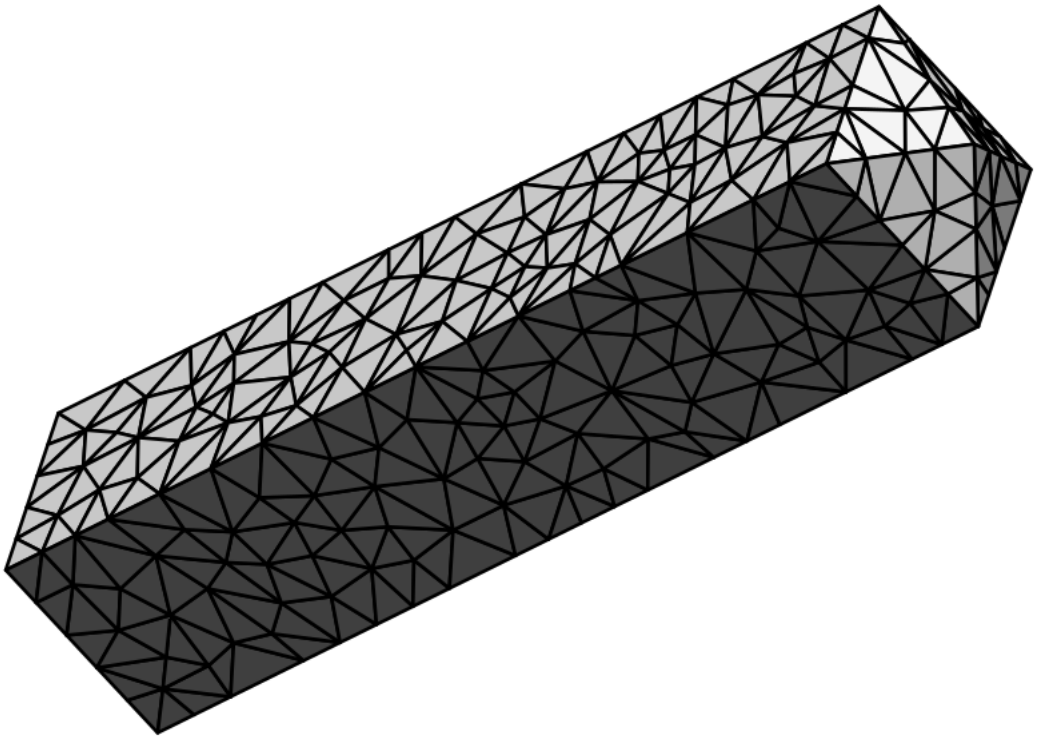
Rysunek 5.1: Example 3D tetrahedron mesh that does not correctly model the domain of the cube. Courtesy of [8]

Second important property is the control of individual elements size. Controlling the volume of elements enables the user to control the accuracy of computations. Bigger number of smaller elements can lead to more accurate but slower computations.

Another problem that has to be solved is the control of elements shape. We often want our tetrahedra to have regular shapes. Meaning that extreme angle values should be avoided. We can ensure the regular shape by controlling the lower bound of the smallest angle in a tetrahedron which also controls the largest angle. Some examples of tetrahedron shapes are shown in figure 5.2.



Rysunek 5.2: Example of some common tetrahedron shapes. Courtesy of [9]



Rysunek 5.3: Example 3D tetrahedron mesh. Courtesy of [8]

Piecewise Linear Complexes

The Delaunay Triangulation

Two-Dimensional Delaunay Refinement

Three-Dimensional Delaunay Refinement

Implementation

The implementation of this thesis has two parts: numerical library and 3D graphic engine. The numerical library contains all the tools to compute real time finite element method solver. The library is then integrated into a demo 3D engine.

RTFEM

RTFEM stands for Real-Time Finite Element Method. The library computes the entire process of FEM for solid mechanics:

1. Pre-Processing: 3D Finite Element Meshing
2. Solver: CPU and GPU dynamic, linear FEM Solver.
3. Post-Processing: Output the displacements.

RTFEM was written in c++11 and uses general template programming. Template programming is used to allow the user to easily change the precision between float and double. RTFEM can be built on any platform supporting cmake. All the tests were made on 64bit Arch Linux. The source code is available in [23].

Folder Structure

RTFEM folder structure looks as follows:

1. /documentation - library documentation
2. /external - all the external libraries
3. /sources - sources code of the library
4. /tests - unittests

For more information about building the library see ReadMe.md in root directory.

External Libraries

RTFEM uses the following external libraries:

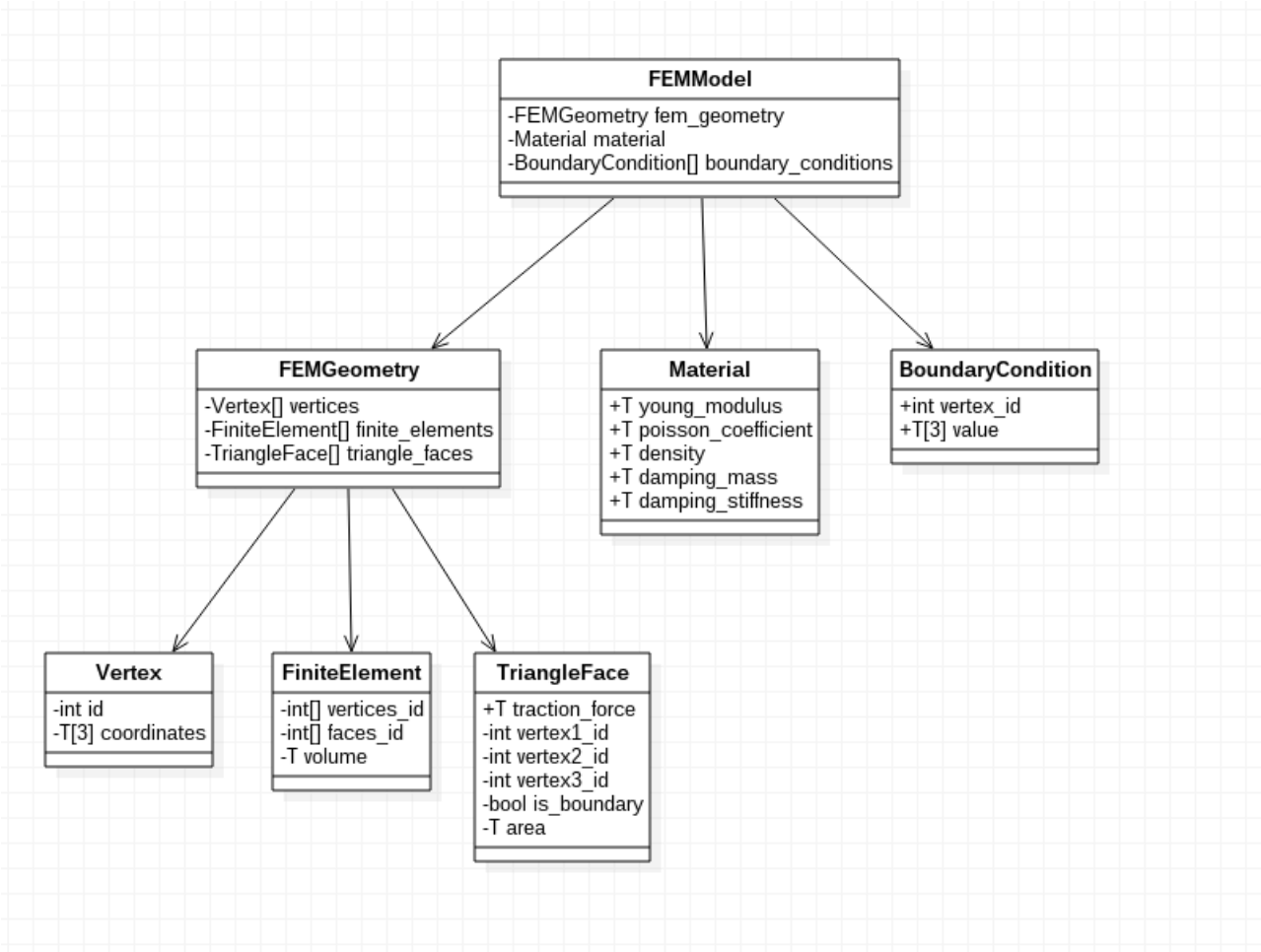
1. TeTGen [9]: 3D Tetrahedron meshing algorithm.
2. Eigen [20]: CPU Matrix operations and System of Linear Equation Solver
3. CUBLAS [18], CUSPARSE [19]: GPU Matrix operations and System of Linear Equation Solver
4. googletest [21]: Unit test framework.

Architecture

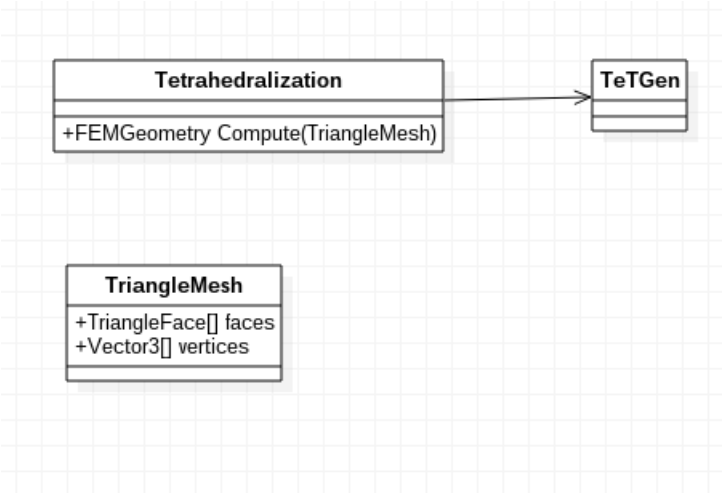
Let us start by describing the architecture of the library. First, the data structures used to represented the FEM model are shown in figure 6.1. FEMModel class is the main container of all sub-components. It also enabled the user to set up body force which acts on all vertices in the model. FEMGeometry holds all the information about geometry of 3D tetrahedron mesh. Vertex is defined by and id and coordinates. TriangleFace is defined by three pointers to vertices. Moreover it contains information about its area, whether it is boundary face and current traction force applied to it. User can apply traction forces directly to the triangle faces. FiniteElement contains pointers to vertices and faces. It also stores information about its volume. Material class is used to store the information of physical material used in the simulation. The variables $damping\_mass$ and $damping\_stiffness$ are used to compute damping matrix C and correspond to α and β from Eq. 2.33. Finally the BoundaryCondition is used to store information about a single boundary condition. User defines the vertex id and the value which this vertex should take.

The post processing part, namely the 3D mesh generation, is mostly done by TeTGen library. The figure 6.2 shows the simply process. TriangleMesh is composed by a set of vertices and faces. Such a triangle mesh can be easily converted from any triangle mesh used in 3D graphics libraries such as OpenGL. Moreover, this triangle mesh is assumed to be a closed mesh. Such a triangle mesh can be easily described as PLC explained in section 5.1. The Tetrahedralization class computes and outputs FEMGeometry.

The solver part of the system is presented in figure 6.3. First, the FEMGlobalDynamicAssembler and TetrahedronLocalAssembler are used to assemble all the required components and store them in FEMGlobalAssemblerData. GPUMVSparseMultiplication is used for the sparse matrix-vector multiplication. GPUSparseCGLinearSolver is the sparse conjugate gradient solver. FEMDynamicSolver class is in the center of the entire solver system. It pre-computes and

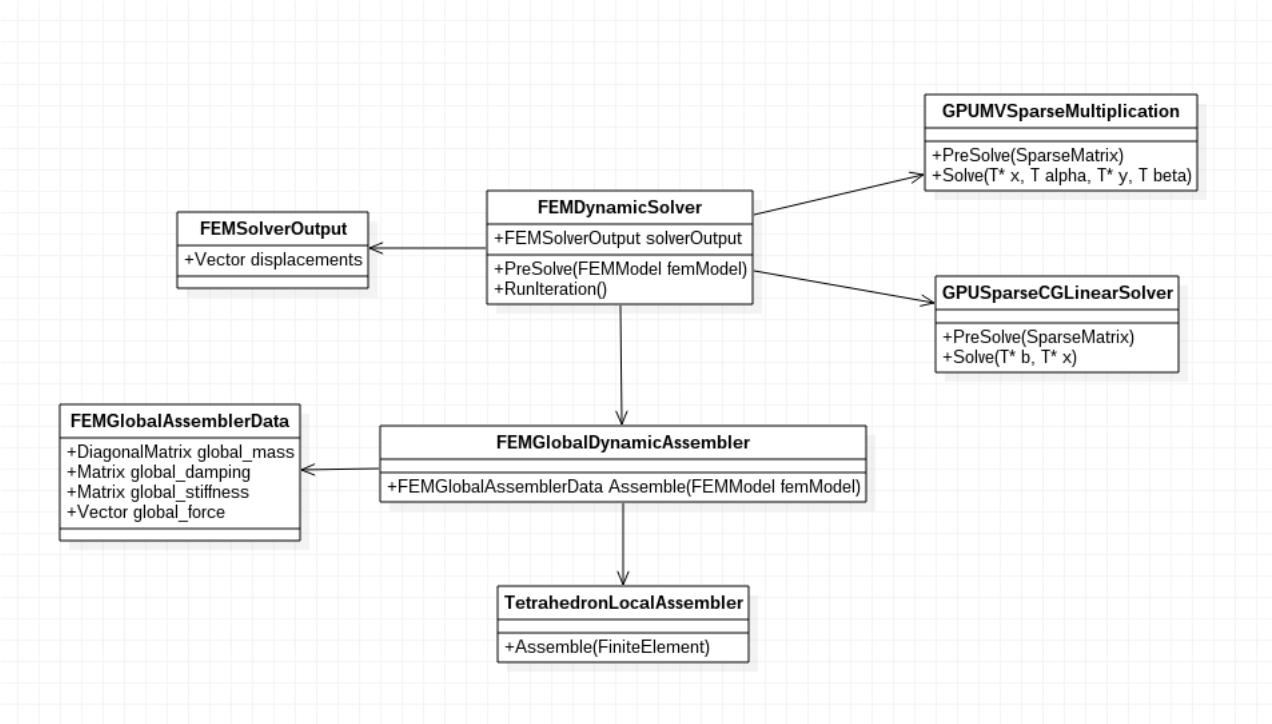


Rysunek 6.1: Class diagram of RTFEM data structures



Rysunek 6.2: Class diagram of RTFEM tetrahedralization process

stores all required data and runs computations for each iteration. The solver uses implicit euler integration method as described in section 2.4.1. The output displacements are stored in FEMSolverOutput class.



Rysunek 6.3: Class diagram of solver

Solver Algorithms

The FEMDynamicSolver has to public functions: PreSolve and RunIteration. PreSolve is used to compute and store all the constant data that are used throughout all iterations. Algorithm 4 shows the necessary steps.

Algorithm 4 FEM Dynamic Solver: PreSolver

```
1: procedure PreSolve
2:     InitAssembly()
3:     InitDisplacementData()
4:     InitPreSolveLHS()
5:     InitPreSolveRHS()
```

InitAssembly algorithm 5 computes the assembly process and stores an instance of FEMGlobalAssemblerData.

Algorithm 5 FEM Dynamic Solver: InitAssembly

```
1: procedure InitAssembly
2:     femAssemblerData = FEMGlobalDynamicAssembler.Compute(femModel)
```

InitDisplacementData simply creates new zero vectors, see algorithm 6.

Algorithm 6 FEM Dynamic Solver: InitDisplacementData

```
1: procedure InitDisplacementData
2:     displacementVelocityCurrent = new Vector()
3:     displacementPositionCurrent = new Vector()
```

InitPreSolveLHS presented in algorithm 7 computes left hand side(LHS) of the system shown in section 2.4.1 and transforms it to sparse CSR format. Then, the boundary conditions are applied but only to matrix LHS, since the force vector will change every iteration. Finally, the sparse matrix LHS is stored on the GPU using the GPUSparseCGLinearSolver class.

Algorithm 7 FEM Dynamic Solver: InitPreSolveLHS

```
1: procedure InitPreSolveLHS
2:     LHS = ComputeLHS()
3:     TransformToSparse(LHS)
4:     ApplyBoundaryConditionsMatrix(LHS, boudaryConditions)
5:     GPUSparseCGLinearSolver.PreSolve(LHS)
```

InitPreSolveRHS in algorithm 8 pre-computes the right hand side vector. It transforms global mass and global stiffness matrices to sparse CSR formats and stores them on the GPU using GPUMVSparseMultiplication class.

Algorithm 8 FEM Dynamic Solver: InitPreSolveRHS

```
1: procedure InitPreSolveRHS
2:     TransformToSparse(globalMass)
3:     GPUMVSparseMultiplicationMass.PreSolve(globalMass)
4:     TransformToSparse(globalStiffness)
5:     GPUMVSparseMultiplicationStiffness.PreSolve(globalStiffness)
```

Computations for each iteration are shown in algorithm 9. ReAssembleForces process, simply gathers the traction and body forces and re assembles them into the force vector. ResetForces, resets them back to zero.

SolveForDisplacements function shown in algorithm 10 computes the implicit euler integration method. First we compute the right hand side vector using the two GPUMVSparseMultiplication classes initiated in the pre solver. We apply the boundary conditions to the RHS. Then we are ready to run the conjugate gradient solver on the GPU to solve our system for velocities. New velocities are used in integration to compute new displacement vector.

Computing right hand side vector and solving linear system of equations require the most computation power. These two functions are fully accelerated on GPU using CUDA.

Algorithm 9 FEM Dynamic Solver: RunIteration

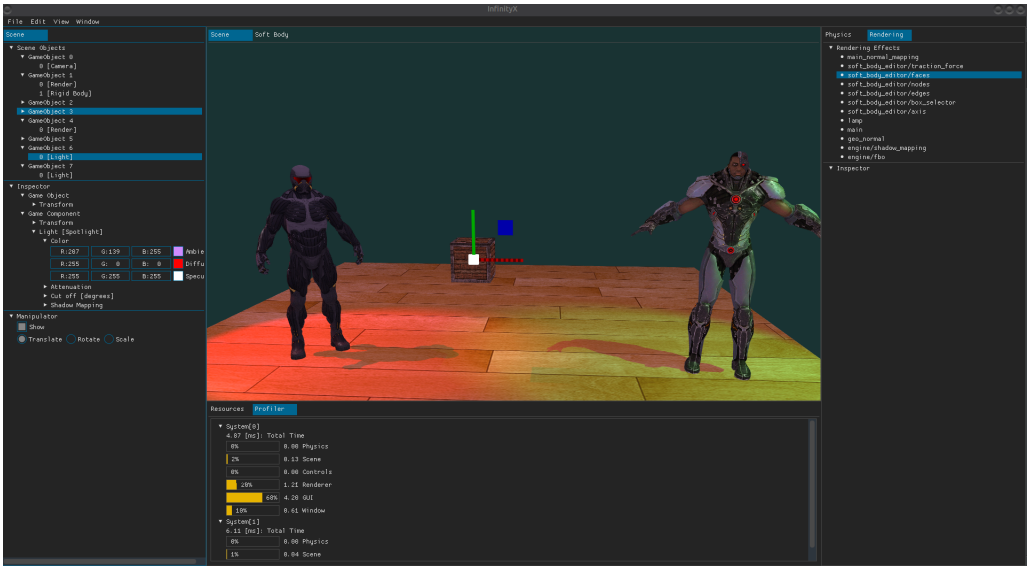
```
1: procedure RunIteration
2:     ReAssembleForces()
3:     SolveForDisplacements()
4:     ResetForces()
```

Algorithm 10 FEM Dynamic Solver: SolveForDisplacements

```
1: procedure SolveForDisplacements
2:     RHS = ComputeRHS()
3:     ApplyBoundaryConditionsVector(RHS, boundaryConditions);
4:     newVelocity = GPUSparseCGLinearSolver.Solve(RHS);
5:     Integrate(newVelocity);
```

RTFEM Integration into Game Engine

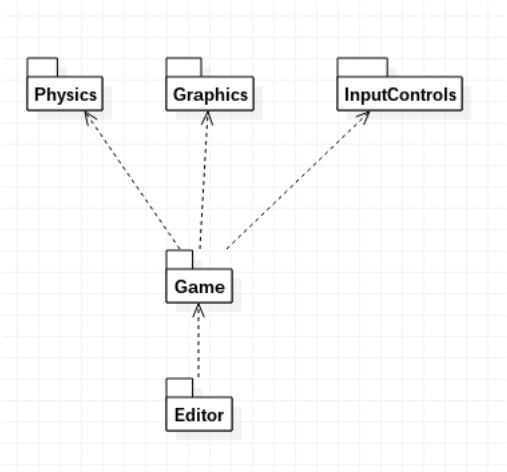
For the purposes of this thesis RTFEM has been integrated into open source game engine created by the author of this thesis, see source code [22]. The entire architecture of the engine is out of the scope of this text. However, the following section will describe the components needed to understand the process of integrating RTFEM. Figure 6.4 shows screenshot from the engines editor.



Rysunek 6.4: Game engine [22]

Figure 6.5 shows the most important modules. Physics module is devoted to run rigid body and soft body simulations. For rigid body simulations it encapsulates lower level rigid body libraries such as PhysX 3.4 [17] and Bullet [14]. Soft body simulation is implemented using

RTFEM. Graphics module is built on top of OpenGL [15]. It encapsulates low level OpenGL programming to increase the productivity. It supports rendering with basic texture materials such as diffuse, ambient and normal mapping. It also supports lighting and shadow effects. User can easily program and edit shader program in run time. InputControls is a module responsible for handling user input from mouse and keyboard. Game module combines all individual modules and provides set of rules for how they should interact with each other. All modules are fully decoupled and communicate with each other through Game module. Finally, Editor module is responsible for graphical user interface, which allows for creating 3D objects, rigid bodies, editing shaders and soft body objects.

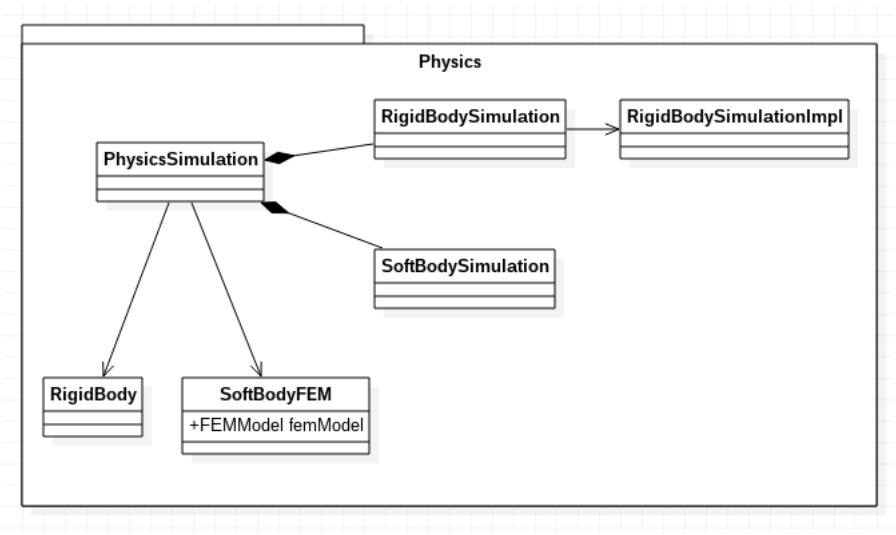


Rysunek 6.5: Game engine modules overview

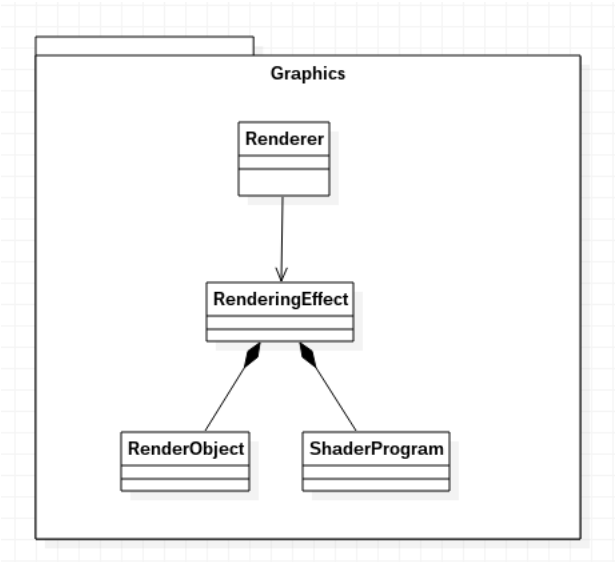
Physics module is described in more detail in figure 6.6. RigidBody and RigidBodySimulation use the bridge design pattern to inject different implementations such as PhysX, Bullet or other libraries that the user might want to implement. SoftBodySimulation uses FEMDynamicSolver to simulate physics for SoftBodyFEM objects.

Most important systems of Graphics module is shown in figure 6.7. Renderer contains RenderingEffects which are composed of RenderObjects and ShaderPrograms. RenderObject encapsulates the triangle mesh and materials such textures and other parameters. ShaderProgram interprets these triangles meshes and materials by programmable shaders using GLSL language.

Basic GameLoop is presented in figure 6.8. The GameContainer contains GameObjects which are composed of GameComponents. GameObject are object that appear in the scene. They can be transformed to any coordinate system by translation, rotation or scaling. We add GameComponents to make the game objects more interesting. Currently the engine supports multiple game components: light source, camera, render object, rigid body or soft body.



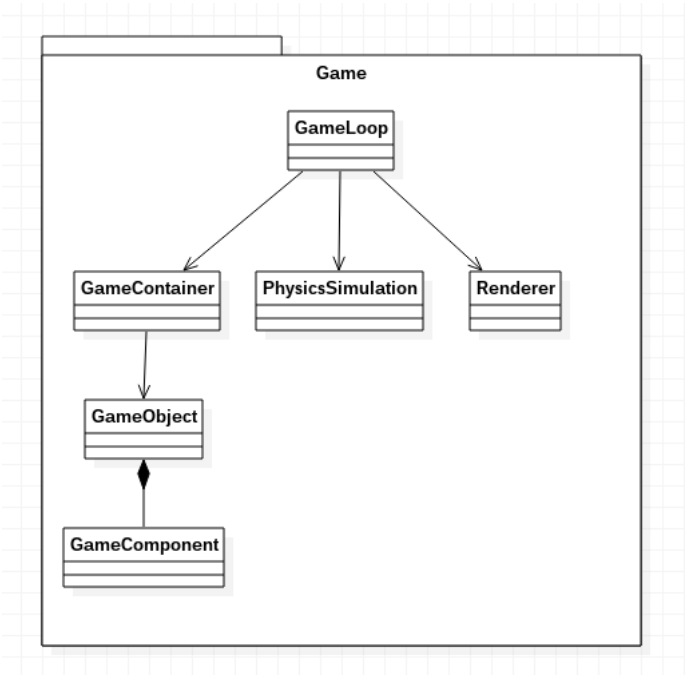
Rysunek 6.6: Game engine physics module



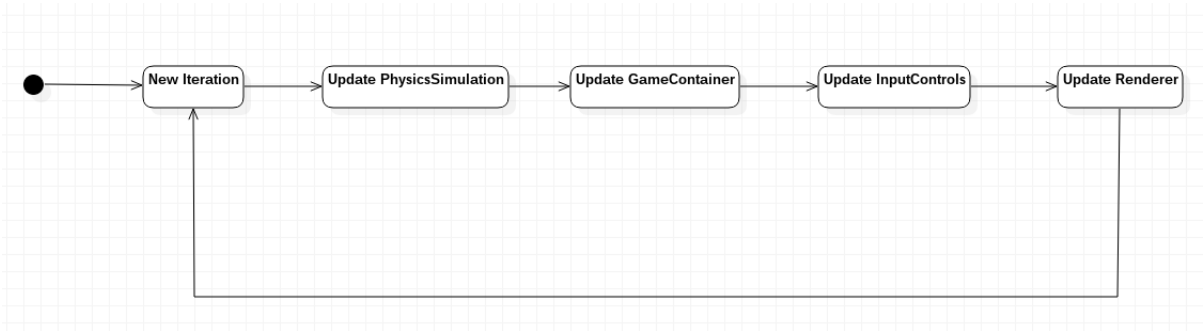
Rysunek 6.7: Game engine graphics module

The game loop sequence is shown in figure 6.9. Each system is updated in a sequence. However each system is updated with different frequency, which can be set by the user. For example, we can update physics twice as often as rendering graphics. GameContainer update is used to synchronize transformations of rigid bodies computed in physics simulation with render components.

Finally, figure 6.10 described the integration of soft body into game module. SoftBodyFEM-Component is created by deriving the GameComponent interface and SoftBodyFEM from the physics module. The soft body component also contains a render object which is created by fetching the boundary triangle faces of FEM model. SoftBodyFEM is updated in PhysicsSimulation which computes and stores the displacements. SoftBodyFEMComponent uses the displacements



Rysunek 6.8: Game engine game module

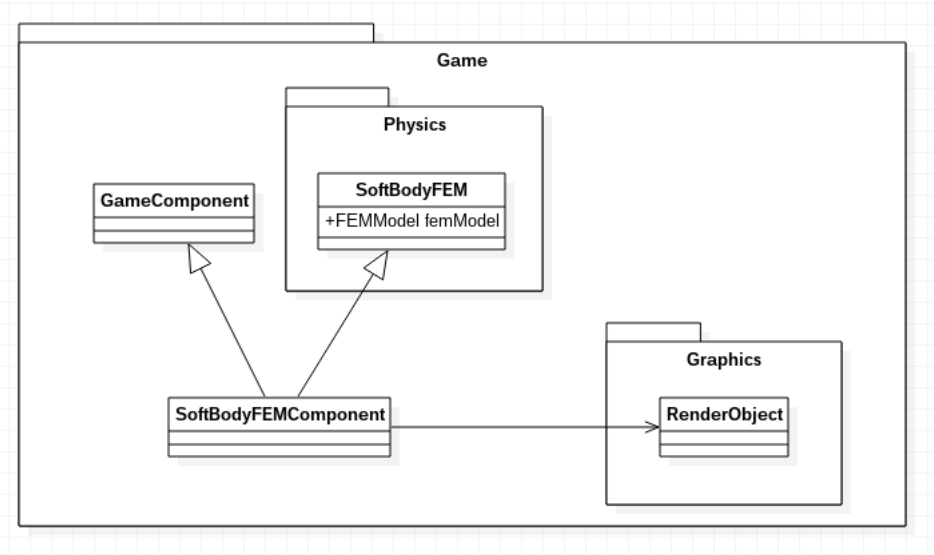


Rysunek 6.9: Game engine game loop sequence

to synchronize render-able vertices by updating the current vertex buffer object(VBO). That happens in the GameContainer update function. Finally the boundary triangle faces are rendered in Renderer update.

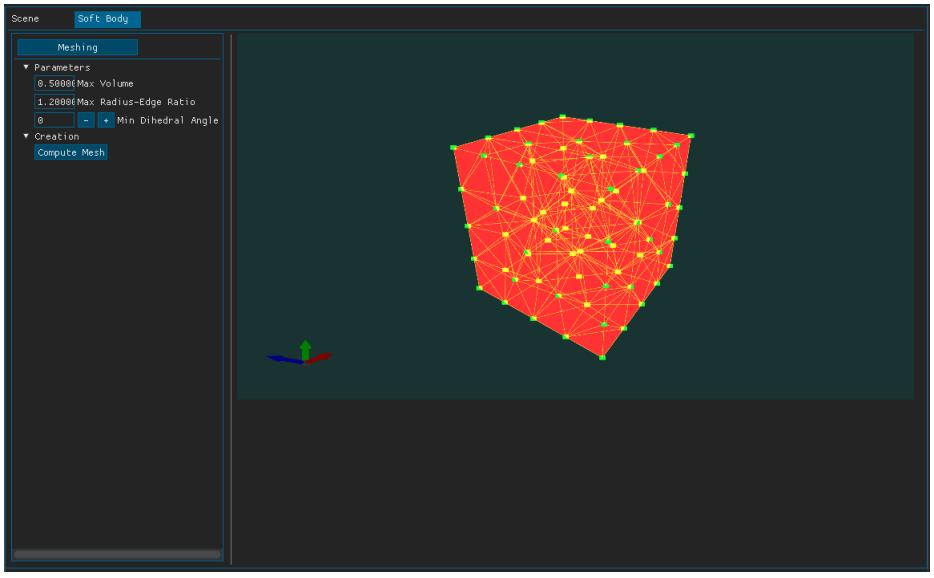
Editor

This section describes the graphical editor of soft body system in the game engine. Soft body editor can be opened by clicking the Soft Body button located in the middle view port, see figure 6.4. We can select a game object to be opened by right clicking it in the scene list(left panel). Game object that can be opened in the soft body editor must have exactly one render component. It can have other components too. The render component contains the triangle mesh that is used as a input problem domain in the 3D mesh generation. Figures 6.11 through 6.16



Rysunek 6.10: Game engine game module. Soft body integration

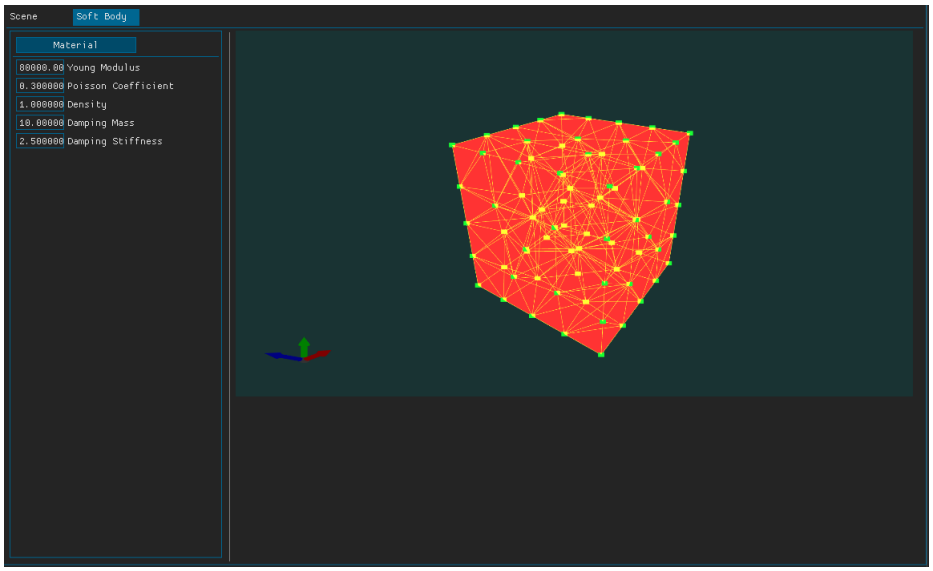
present the soft body editor panels. In the first step we want to create 3D mesh. Meshing panel is devoted to that procedure. Figure 6.11 shows control parameters explained in chapter 5. The cube has been already meshes with example parameter values.



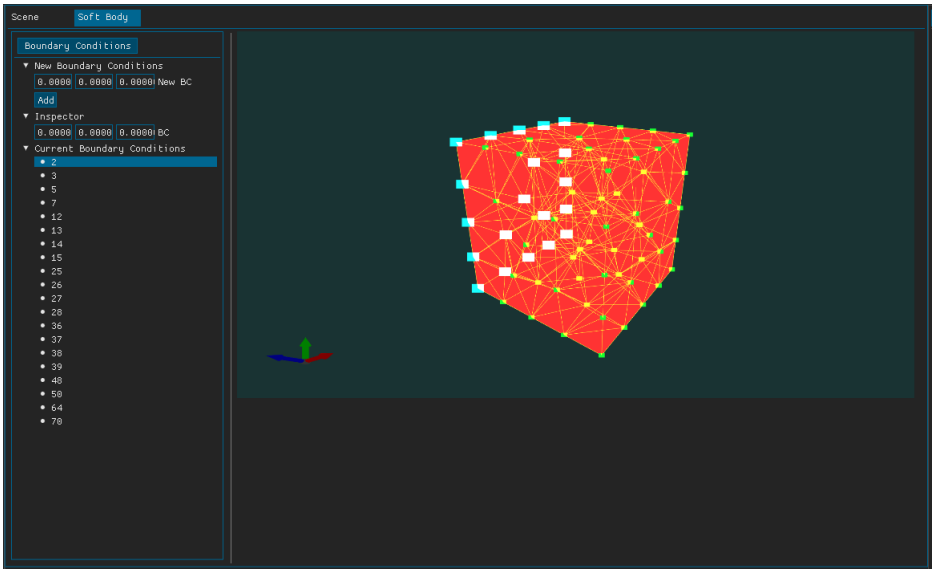
Rysunek 6.11: Soft body editor. Meshing panel

Next, figure 6.12, we are presented with materials described in chapter 2.

Boundary conditions panel is shown in figure 6.13. The user can select multiple vertices by holding down the left mouse button. Then the user inputs the value for new boundary conditions. In this example, the velocity of selected vertices will be equal to zero. Thus, the vertices will stay put.



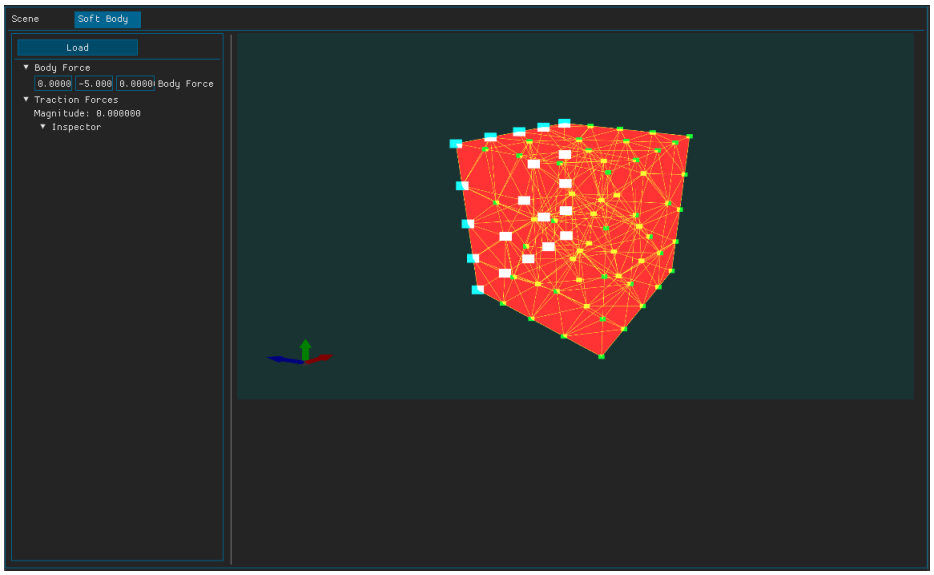
Rysunek 6.12: Soft body editor. Meshing panel



Rysunek 6.13: Soft body editor. Boundary conditions panel

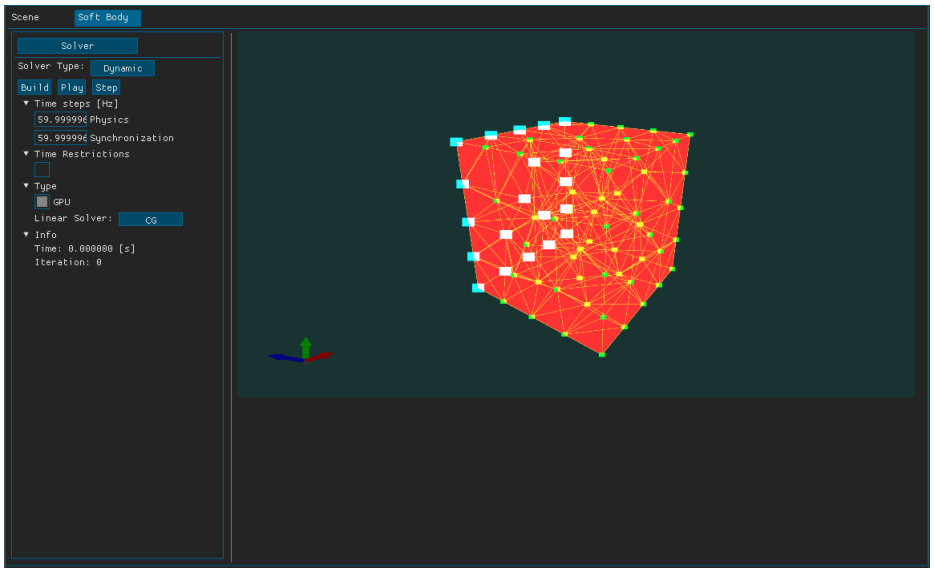
In the load panel(see figure 6.14 user can select body force that acts on the entire body, simulating gravity. Traction forces are selected by left clicking on a triangle face and dragging the mouse to choose the magnitude of traction force. Remember that the direction of the traction force lies along the normal of that face.

Finally, the solver panel is shown in figure 6.15. Build button assembles the model using the geometry and parameters chosen in previous panels. Play button starts the simulation. Step button steps the simulation by a single iteration. We can choose the frequency of physics simulation and the synchronization of output displacements with vertex buffer object. Time restrictions can be selected to stop the simulation after input amount of seconds. We can select if the solver

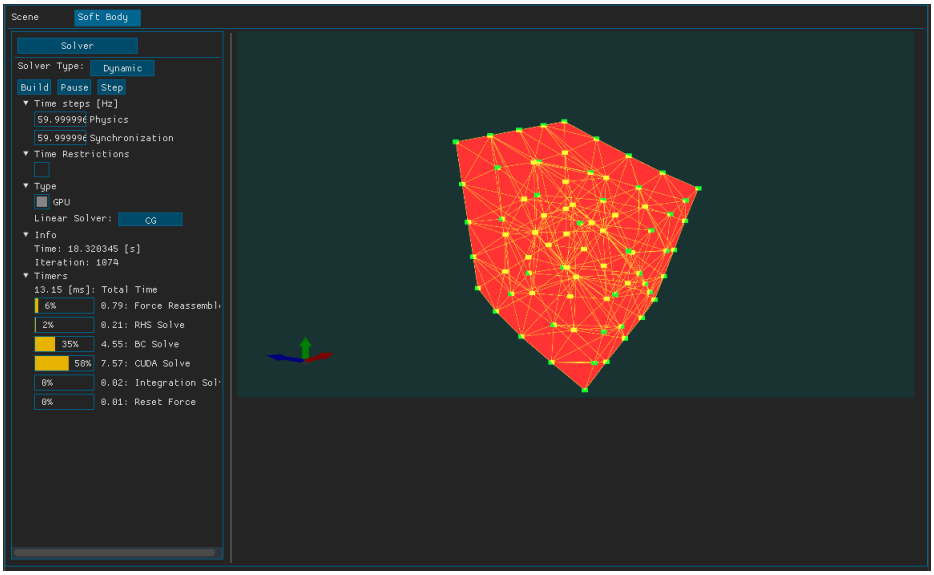


Rysunek 6.14: Soft body editor. Load panel

should be computed on GPU or CPU. Linear solver can be chosen between Conjugate Gradient and LU decomposition. Figure 6.16 shows the same model after about 20 seconds of simulation. The cube has been deformed under the gravity force. The timers are uses to profile the solver algorithms. It shows the time of computations for the last iteration.



Rysunek 6.15: Soft body editor. Solver panel



Rysunek 6.16: Soft body editor. Solver panel during computations

Tests

Conclusions

The proposed real-time FEM solver library has been integrated into existing game engine. The library runs in parallel with existing physics and graphical modules. The two modules are fully decoupled and communicate only through a common game modules. Such architecture supports cheaper maintenance of entire system and makes it easy to inject other soft body libraries. The high level abstraction of the library reflects process of designing a FEM model. Thus, it was trivial to create a visual editor in which the user can design and solve FEM models. GPU solver has obviously been proven to run much faster than a CPU for a large enough vertex count.

Further work

The two most expensive procedures of solver iteration include computing the right hand side vector and solving linear system of equations. Those procedures have been fully accelerated on the GPU. Other procedures such as applying boundary conditions or computing the force vector are now causing bottleneck due to their CPU implementation. Further work should look into accelerating these procedures on the GPU side as well.

Soft body deformation based on FEM formulation can support fracture mechanism. Stress magnitude can be calculated at each vertex. After overcoming a certain yield threshold the material would start breaking.

The proposed algorithm used a linear formulation of solid mechanics. This can only model small deformations. Next step of research should include study of GPU accelerated non-linear solvers. Such solvers could implement more accurate formulation for solid mechanics. Moreover, the solver could also include solutions for heat transfer problems and the coupling between the thermal and solid mechanics. Such solution would improve accuracy and provide even more realism to virtual environments.

It would be interesting to see how the proposed soft body solution checks out with virtual reality hardware. The user would be able to deform a body by applying pressure using their hand

controller. Moreover, soft body could be grabbed by any of the boundary vertices. Such a user interface could be a great realism test.

Bibliografia

[1] Michał Kleiber, Piotr Kowalczyk, *Wprowadzenie do Nieliniowej Termomechaniki Ciał Odkształcalnych.*

[2] Zienkiewicz *The Finite Element Method. Volume 1: The Basis.*

[3] Zienkiewicz *The Finite Element Method. Volume 2: Solid Mechanics.*

[4] Altair University *Practical Aspects of Finite Element Simulation.*

[5] Department of Aerospace Engineering Sciences University of Colorado at Boulder *Advanced Finite Element Methods (ASEN 6367) Lectures https://www.colorado.edu/engineering/CAS/courses.d/AFEM.d/.*

[6] Matthias Muller, Jos Stam, Doug James, Nils Thurey, *Real Time Physics Class Notes.*

[7] Jonathan Richard Shewchuk *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.*

[8] Jonathan Richard Shewchuk *Lecture Notes on Delaunay Mesh Generation.*

[9] Hang Si *A Quality Tetrahedral Mesh Generator and 3D Delaunay Triangulator.*

[10] Miles Macklin, Matthias Muller, Nuttapong Chentanez, Tae-Yong Kim *Unified Particle Physics for Real-Time Applications.*

[11] Eric G. Parker, James F. O'Brien *Real-Time Deformation and Fracture in a Game Environment.*

[12] X. Faure, F. Zara, F. Jaillet, J-M. Moreau *An implicit Tensor-Mass solver on the GPU for soft bodies simulation.*

[13] Yan Zhuang, John Canny *Real-time Simulation of Physically Realistic Global Deformation.*

[14] Bullet *http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Documentation.*

[15] OpenGl *https://www.opengl.org/.*

[16] SOFA Framework *https://www.sofa-framework.org.*

[17] NVIDIA *http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Index.htm*

[18] NVIDIA cuBLAS *http://docs.nvidia.com/cuda/cublas/index.html.*

[19] NVIDIA cuSPARSE *http://docs.nvidia.com/cuda/cusparse/index.html.*

[20] Eigen *http://eigen.tuxfamily.org/index.php?title=Main_Page.*

[21] Google *https://github.com/google/googletest.*

[22] Jakub Ciecierski *https://github.com/Jakub-Ciecierski/InfinityXLib.*

[23] Jakub Ciecierski *https://github.com/Jakub-Ciecierski/RT-FEM.*

[24] Jakub Ciecierski, *https://github.com/Jakub-Ciecierski/SoftBodySimulation*

Wykaz symboli i skrótów

FEM	Finite Element Method
GPU	Graphics Processing Unit
CPU	Central Processing Unit
PDE	Partial Differential Equation
ODE	Ordinary Differential Equation

Spis rysunków

Spis tabel