Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Simulator of Unicellular Microorganism

Kristian Senčuk

Supervisor: Ing. Mgr. Marek Dvorožňák
Field of study: Open Informatics
Subfield: Computer and Information Science
May 2018

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Senčuk Kristian**	Personal ID number:	**456905**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Branch of study:	**Computer and Information Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Simulator of Unicellular Microorganism

Bachelor’s thesis title in Czech:

Simulátor jednobuněčného mikroorganizmu

Guidelines:

1. Familiarize yourself with the implementation of a suitable 2D rigid body and fluid simulation library such as [1].
2. Refer to the videos of the microscopic world [2, 3] and design and implement a simulator of a generic unicellular microorganism that is able to move and interact in 2D environment including the ability to absorb surrounding objects. The solution must be able to perform real-time simulation and must be visually faithful to the real-world videos.
3. Create at least two specializations of the generic microorganism that differ for instance in rigidity of its body and also in visual appearance.
4. Utilize the simulator to create a prototype of a game in the style of [4] where the player increases score by controlling the microorganism to consume objects in its surroundings.

Bibliography / sources:

[1] Google - LiquidFun - 2013, http://google.github.io/liquidfun/
[2] Rogers David - Crawling Neutrophil Chasing a Bacterium - 1950s,
https://www.youtube.com/watch?v=I_xh-bkiv_c
[3] Staifan Basel - White blood cells under the microscope 2000X - 2016,
https://www.youtube.com/watch?v=sYCUWqc_x3U
[4] Valadares Matheus - Agar.io - 2015, http://agar.io/

Name and workplace of bachelor’s thesis supervisor:

Ing. Mgr. Marek Dvorožňák, Department of Computer Graphics and Interaction, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **08.01.2018** Deadline for bachelor thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

Ing. Mgr. Marek Dvorožňák	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

Acknowledgements

I would like to thank my supervisor Ing. Mgr. Marek Dvorožňák for his assistance and practical advice regarding this thesis. I would also like to express my gratitude to the Department of Cybernetics for allowing me to work on this custom topic. Last but not least, I thank my family and friends for their constant support.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 21. May 2018

. . . .
Signature

Abstract

This bachelor thesis engages the problem of simulating unicellular microorganisms in real time. It introduces the reader to existing solutions and describes the innovative methods used to implement a playable simulation that outperforms its competition in terms of visual resemblance to real life reference material. The dominant component of this work is the application showcasing the implemented simulation features.

Keywords: simulation, cell biology, Box2D, microscope

Supervisor: Ing. Mgr. Marek Dvorožňák
FEE, Department of Computer Graphics and Interaction

Abstrakt

Tato bakalářská práce se zabývá problematikou real time simulací jednobuněčných mikroorganizmů. Seznamuje čtenáře se stávajícími řešeními a popisuje inovativní metody použité při implementaci hratelné simulace, která překonává konkurenci z pohledu vizuální podoby realistickým předlohám. Dominantní částí této práce je aplikace, která demonstruje implementované funkční prvky v simulaci.

Klíčová slova: simulace, biologie buňky, Box2D, mikroskop

Překlad názvu: Simulátor jednobuněčného mikroorganizmu

Contents

Figures

Tables

Chapter 1

Introduction

The aim of computer simulations is to reproduce the behavior of observed natural systems by using mathematical models. Such simulations are plentifully applied as tools in many scientific fields, such as physics, chemistry, meteorology or biology. The purpose of such reproductions varies. Some simulations serve as a way of accurately predicting impending occurrences, such as the weather forecast. Others are used to support research and securely test inventions before they are applied in real life situations.

From the point of view of a casual user, a simulation becomes particularly interesting when a visualization program is implemented on top of its computational logic. In such cases, the application can go beyond the scope of science and acquire an educational or a recreational function (i.e. video games).

A subset of biology-themed simulations deals with imitation of microscopic life. Although it does not lack its uses in scientific research, it is mostly demanded in the video game industry. Players from all over the globe have shown great interest in the subject when the video game Agar.io [4] was released. This game, which will be further described in Section 2.2.1, features a playable generic unicellular microorganism which can move around in the game's environment and increase score by interacting with its surroundings.

Agar.io became an instant hit amongst casual players. In 2015, the game's name was the third most searched keyword on Google worldwide [5]. The decisive factor in its success is debatable. Many reviews, such as the one from technology blog Engadget [6], credit it to the game's unusual and interesting concept of natural selection in a microscopic environment, as depicted in Figure 1.1.

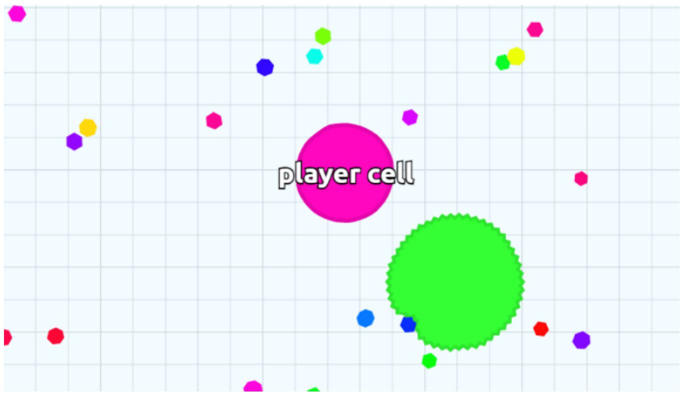


Figure 1.1: In-game view of Agar.io

It is apparent that Agar.io has become closely associated with this topic, despite many flaws in its visual fidelity. Players deduce the connection between the game and its theme primarily from its gameplay, not aesthetics. Just by observing motionless screenshots of the game, it is difficult to accurately recognize the resemblance to microscopic environment.

To this date, a lot of room is left for visual improvements and innovations in this regard. This thesis sets its goal to design and implement a simulator of the most complex and visually distinct actor in the microscopic world - the unicellular microorganism, as well as its interactive environment. Put simply, the resulting simulation must output a view that is visually comparable to what can be seen under the microscope in real life. Needless to say, the simulation will run in real time and thus provide broad interactability with user input. As the highlight of this work, a game concept similar to Agar.io will be devised in an attempt to show the visual superiority to its existing counterparts.

Note that the key element in this work is visual resemblance to real life references, namely from the perspective of a person unrelated to professional biology.

Chapter 2

Background

2.1 Cellular biology

A cell is the fundamental structural and functional unit of a living organism. Its dimensions usually range from 1 to 100 micrometers [7]. Thus, most cells cannot be seen without a microscope.

From a scientific standpoint, cells can be divided into many classification groups based on numerous criteria. The most apparent criterion observes whether the cell functions as an independent organism on its own, or is part of a greater aggregate. The former is known as a unicellular organism, while the latter constitutes a multicellular organism.

2.1.1 Multicellular organisms

Multicellular organisms consist of multiple cells acting in unison. Animals, plants and most fungal species are all multicellular organisms. Due to the cells being bound to each other, not much mobility can be observed under the microscope. An example of such organism is depicted in Figure 2.1.

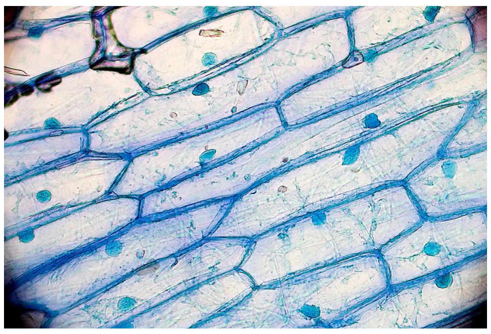


Figure 2.1: Onion cells under a microscope [1]

2.1.2 Unicellular organisms

Unicellular organisms consist of only one cell. They are much more mobile than their multicellular counterpart. Two types of movement can be observed in such organisms.

Cilia- and flagella-powered movement

This type of movement involves auxiliary structures known as the cilium (found in parameciidae) or flagellum (found in many kinds of bacteria or animal spermatozoon), both of which are depicted in Figure 2.2. These structures propel the mass of the cell body in a fluid environment, causing only a slight deformative effect on the cell body.

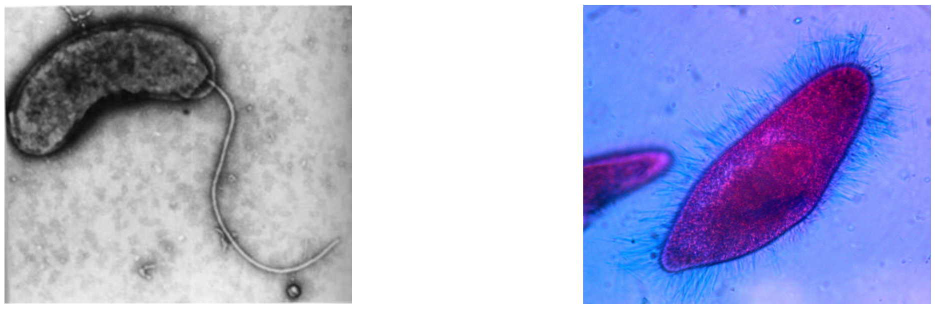
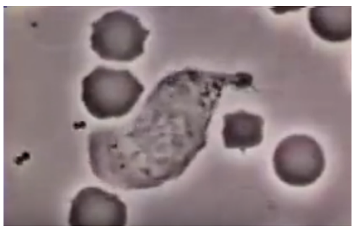

(a) : Bacterium with one flagellum [8] **(b) :** Paramecium with multiple cilia [9]

Figure 2.2: Examples of microorganisms with cilia- or flagella-powered movement

Amoeboid movement

The second type – amoeboid movement, is much more visually distinct. The methods behind it are related to complex chemical reactions, the details of which are out of this work's scope. Simply put, it involves an inner high-density fluid called the cytoplasm that causes the cell to crawl as a result of its protrusion. Such cells have very deformative properties, as seen in amoebas or white blood cells (see Figure 2.3).

In this work, a special type of white blood cells called the neutrophil will be used as a central reference for unicellular organisms with amoeboid movement. The alternative cilium-based movement will be modeled after parameciidae.


(a) : Neutrophil cell [10]

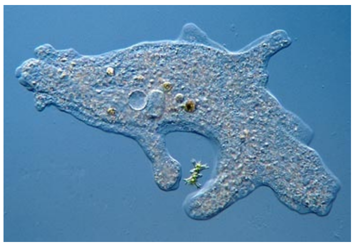

(b) : Amoeba cell [11]

Figure 2.3: Examples of amoeboid unicellular organisms

2.2 Existing solutions

A number of existing solutions engaging the subject of microorganism simulation can be observed in various fields of study. The following subsections will list and describe the most distinct ones used for educational, recreational and scientific purposes.

2.2.1 Video games

Spore

Spore [12] is a commercial video game developed by Maxis. It is centered around the idea of organism evolution.

While not focusing on the microscopic world alone, the game's first phase features a user controllable unicellular microorganism which can move in its environment, consume nourishment and fight other organisms. Note that the game is heavily stylized - the cells have a rather comical appearance (see Figure 2.4). It does not attempt to simulate real-life references.

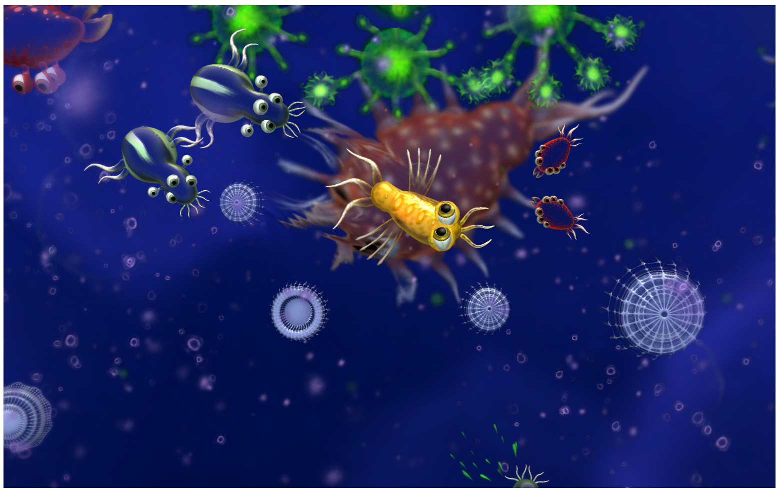


Figure 2.4: In-game view of the cellular phase in Spore

Due to the commercial nature of the game, the methods used in the cell's body implementation are not open to the public. It is clear, however, that the cell's body is three-dimensional and does not seem to be physics-based. The movement and consumption of nourishment is most likely strictly animated.

Although the game's microbiological principles were often criticized to be highly inaccurate, the playable microscopic simulation in Spore remains one of the most memorable and advanced ones, as well as the only one developed with a high budget.

Osmos

In 2009, Osmos [13] by Hemisphere Games was released for desktop and mobile platforms. The game took the "survival of the fittest" concept in the context of microscopic life from Spore and kick-started a whole genre based on it. The goal of this simple game was to absorb other organisms and consequentially become larger. Osmos was marketed as a relaxing, single player puzzle game.

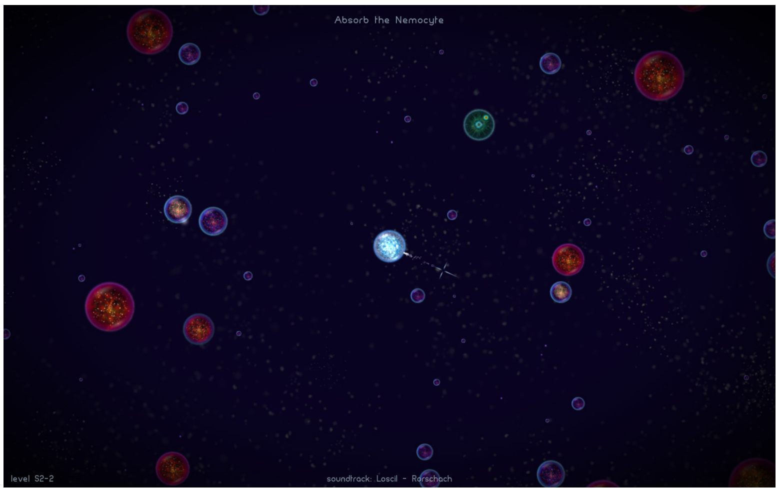


Figure 2.5: In-game view of Osmos

Although the visuals are minimalistic in their essence, they attempt to achieve some degree of resemblance to real-life references by using special effects and animations. The style is no longer comical like in Spore. It instead pursues a scientific look inspired by cosmos (see Figure 2.5). Objects representing unicellular organisms have a slightly distorted circular shape and don't seem to be collision-reactive.

Agar.io

In 2015, Agar.io [4] developed by Valadares Matheus was released for web browsers. Featuring similar concepts to the ones used in Osmos, it further extended its gameplay mechanics to be compatible with online multiplayer. Once again, the player controlled a unicellular organism. Its purpose was to absorb and consume small circular objects known as agar, inspired by a real life biochemical substance of the same name. The more absorbed agar has been gathered, the larger the cell. This time, however, competitive elements were added, and the player's larger cell could absorb other players' smaller cells.

In this game, a simple soft-body model was used to represent a unicellular microorganism. The model is somewhat physics-based and therefore manages collision interactions with other objects in the simulation (see Figure 2.6). The game's many features, such as absorption, are not physics-based, however.

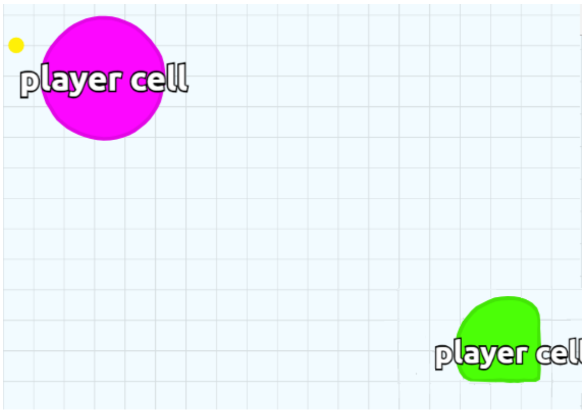


Figure 2.6: Deformative properties of the soft-body (bottom right) in Agar.io

In consideration to the target platform, the style of the simulation is very simplistic and does not include many effects, as opposed to Osmos.

Cellcraft

Cellcraft [14] is an open-source Flash game primarily aimed at students. It serves an educative and motivational purpose to raise interest in cellular biology. It does not put emphasis on gameplay mechanics, instead, it attempts to simulate real life references primarily from a visual point of view.

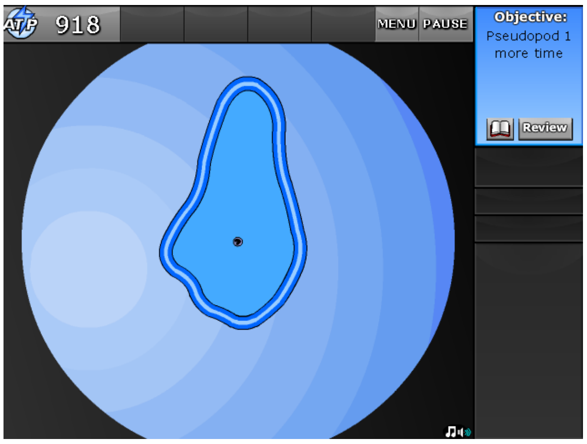


Figure 2.7: Unicellular organism in Cellcraft

The game does not utilize any physics engine and has very limited collision managment. The cell itself, however, is modeled to be much more deformative (Figure 2.7) and to achieve a higher degree of resemblance to real-life amoeboid cells than the aforementioned games. Its body model consists of numerous ordered vertices which define a closed area.

2.2.2 Scientifically precise solutions

Typical tasks in bioinformatics include simulations of complex biochemical reactions. In terms of cellular biology, the dominant subjects of scientific simulation include manipulation with RNA/DNA, methods of nourishment consumption and cell division.

In the research paper *Mechanics of neutrophil phagocytosis: experiments and quantitative models* [15] published in the Journal of Cell Science, a quantitative model was devised for a relatively accurate simulation of phagocytosis. The resulting simulation accounts for realistic biochemical factors, such as protein coagulation.

The movement of amoebas was the subject of study in the research paper *Non-Brownian dynamics and strategy of amoeboid cell locomotion* [16]. Besides using complex biochemical models, the work also researched and consequentially simulated the strategy behind suitable protrusions of the cell body.

Note that scientific simulations do not fit this work's objective. Precise solutions require computations that are heavier on performance, not to mention the implementation itself is expected to be much more difficult. For a common user, most of the scientifically precise features will remain unnoticed. Thus, inspiration will be taken only from visually distinct factors.

2.3 Suitable libraries

The simulation will use certain free open-source instruments to implement various physical models and their respective visual representation. The libraries Box2D, LiquidFun and LibGDX have been chosen to power the simulation.

2.3.1 Box2D

Box2D [17] is a C++ based two-dimensional physics engine library developed by Erin Catto. With its initial release dating back to 2007, it has gone through numerous revisions and ports to other programming languages since then. The library is supported by many popular game engines and frameworks, including Unity3D [18], LÖVE [19], GameMaker: Studio [20], Cocos2d [21] and LibGDX [22].

For the purpose of this project, a LibGDX supported Java wrapper around the native C++ Box2D code will be used.

Features

Box2D offers a wide variety of features in the fields of simulating two-dimensional kinematics and dynamics. Structurally, the library is composed of three complex modules: Common, Collision and Dynamics [23]. The first one serves as a utility encapsulation, general functionality such as memory allocation and various mathematical calculations can be found there. The collision module serves the sole purpose of managing shapes and their interaction with the environment. Finally, the dynamics module utilizes the former modules and defines key objects like the world, body, fixture and joint.

World. The world is a heavy object holding a set of bodies and joints capable of interacting with each other.

Body. A body is a container of physical properties unrelated to collision that affect the body‘s position and velocity. Such properties include its general physical behavior (static, dynamic or kinematic), its ability to rotate, linear and angular velocity modifiers in form of damping and lastly, the bullet attribute which optimizes the body expected to move at very high velocities.

Fixture. Fixtures act as a link between a body and its shape. They also define the shape‘s collision-related properties, such as friction and restitution. Since mass is distributed along the shape of the body, Box2D accounts for it with density – another fixture parameter. The final useful feature of fixtures is collision filtering which allows the programmer to disable collision between certain fixtures.

Joint. Joints define a relation between two bodies or a body and an arbitrary point in the world. A wide variety of joint types is implemented in Box2D. This work will only cover a selection of them that are applicable in the project. One of the most commonly used joints is the distance joint. It implements the relation of maintaining a constant length between two bodies. Note that in case of extreme velocities acting on one of the connected bodies, the length might temporarily extend beyond the set constant. A distance joint can also be made to have spring-like qualities by setting its frequency and damping ratio constants to non-zero values.
The rope joint is another joint type that puts a constraint on the distance between two bodies. In this case, however, a maximum length can be set to prevent it from stretching beyond a limit. Note that while similar, it is not an extension of a distance joint, as it does not attempt to keep a fixed distance between the connected bodies.

Alternatives

Original C++ based Box2D is the basis for numerous ports to other programming languages. One of the more distinct ports is jBox2D [24] – a Java based implementation of this library. While having a generally lower performance than the C++ counterpart, it extends the original library's functionality by adding a number of exclusive features. One of them is the implementation of the so-called constant volume joint, which spans across a closed string of bodies. As the name suggests, the distances between the bodies are adjusted in order to preserve the constant volume of the closed string‘s interior, as seen in [25]. Such feature could prove useful in simulating cells with amoeboid movement, but the performance issues outweigh its benefits.

Besides Box2D, there are other 2D physics engines that compete with it. One of them is the C-based Chipmunk2D engine [26] developed by Scott Lembcke. Since not as many game engines and frameworks support this physics engine, it is not as widely used as Box2D. However, the latest release shows performance superiority over Box2D. Feature-wise, both engines are comparable.

2.3.2 LiquidFun

LiquidFun [27] is a Box2D extension developed by Google used for fluid physics simulation. Its main component is the particle module which is fully compatible with Box2D‘s base modules. In the context of this additional component, a particle is a round atomic unit of the simulated fluid. Multiple particles can be clustered in particle groups. Particles with common properties reside within a single particle system.

Similarly to Box2D bodies, parametrized physical attributes can be assigned to an individual particle or a particle group. LiquidFun features several presets that define the general behavior of particles. For example, the water particles preset is designed to provide a physical resemblance to the properties of water. Elastic particles, on the hand, are closely bound together to form a soft, gelatinous-like body. There is a total of 12 presets that can be tried, most of which are further parametrized. For the purpose of this simulation, only water particles will be used.

LibGDX does not support LiquidFun by default. Therefore, an external LibGDX port [28] must be included to utilize its features.

2.3.3 LibGDX

LibGDX is a free, open-source framework designed to aid programmers in the development of desktop and mobile applications. It is mostly used as a game-development tool. Although it is primarily written in Java, certain sections are natively written in C or C++ to improve performance.

This framework offers certain mathematical utilities and simplifies work with audio, user input and most importantly, graphics. Note that graphics handling happens on a relatively low-level. Therefore, basic knowledge of OpenGL[29] is expected.

One of the most apparent advantages of this framework is its compatibility between desktop and mobile builds. The same core code can be effortlessly interpreted by both platforms.

Chapter 3

Solution and its methodology

3.1 Physical models

3.1.1 Soft-body model

A soft-body refers to a physical model often used to simulate bodies with slight deformative properties. It is based on the idea that the relative distance between the body's local center and any point on its boundary can be dynamically adjusted, as opposed to a rigid body.

A cell simulated by such a model is expected to have strict limitations to what shapes it can appear in. The general shape of the cell will not change, but only slightly deviate from it. Therefore, objects modeled in accordance to this design resemble real life cells with rigid-like properties, such as parameciidae (Figure 2.2.b), which will further act as a real life reference in our attempt to simulate it.

Structure

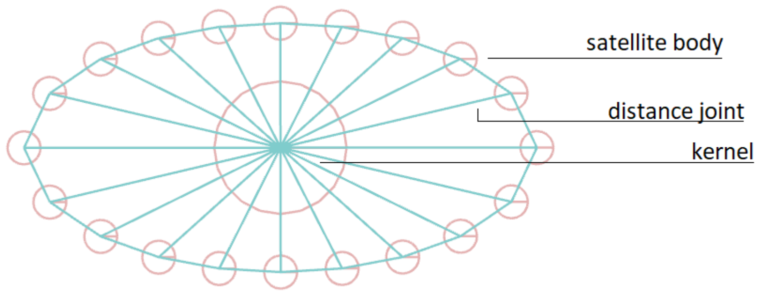


Figure 3.1: Single-layered soft-body structure

The general structure of a soft-body is depicted in Figure 3.1. Acting as the center of a soft-body is the kernel – a circle shaped rigid body. One of the distinct properties of the kernel is that it holds most of the soft-body's mass, so it is appropriate to assign a high density value to it. The kernel is then connected to the satellites - evenly distributed rigid bodies of the same shape which are surrounding the kernel. Satellite bodies constitute the soft-body's dynamic boundaries.

To ensure the dynamic attribute of the boundaries, the connection between the kernel and its satellites must possess a spring-like property. In terms of Box2D, we can achieve this by implementing such a connection as a distance joint with a non-zero frequency and linear damping. It is also necessary to bind neighboring satellite bodies to each other in order to bring stability and closure to the dynamic soft-body shape. For this purpose, an ordinary distance joint with default parameters is fitting.

A soft-body can be further augmented by adding another layer of satellites to improve stability (Figure 3.2). A single layer of satellites may allow the kernel to get outside the closed area – a situation that should be avoided at all costs in our simulation.

Structure

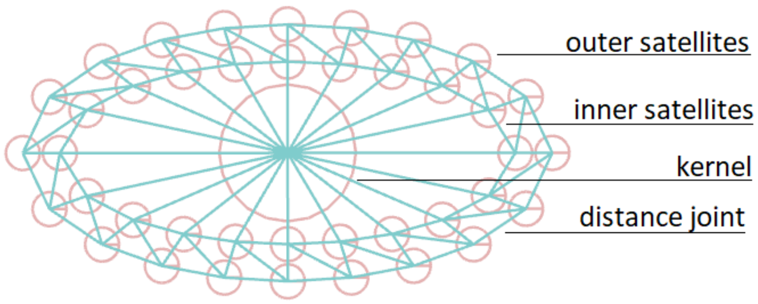


Figure 3.2: Dual-layered soft-body structure

The second layer will be created in the same manner as the first one. To connect the layers, respective bodies in the inner and outer layer must be bound together. However, to fully utilize the extra layer, cross connections will be created for the purpose of ensuring that the respective bodies in opposite layers maintain their relative position.

Movement

By referring to the microscopic footage of parameciidae [30], several properties can be noticed when observing the cell's movement. The cell possesses a front part and it always moves in the direction it is facing. Since the cell is expected to change its direction during the simulation, our implementation must support simple rotation of the soft-body.

First, we assign a single body in the outer-most satellite layer to serve as the front part. When the soft-body receives a command to start moving towards a certain point in the Box2D world, we simply calculate the difference vector between that point and the front body. After normalizing it, we use this vector as the velocity that will be applied to the front body (Figure 3.3).

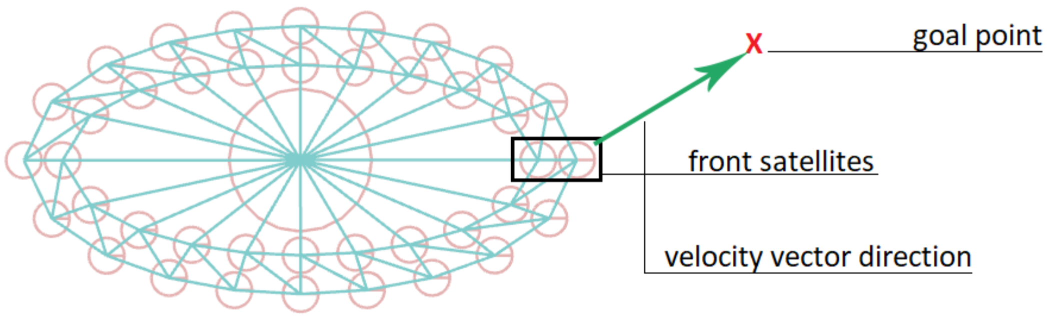


Figure 3.3: Soft-body front assignment and rotating movement

Besides bringing the soft-body in motion, the described method also causes it to rotate until it faces the direction of its goal. An exception to this method will occur if it turns out that the goal point is closer to the front body‘s counterpart on the opposite side of the entire soft-body. Such situation may happen if the soft-body attempts to rotate over 90 degrees. In such cases, it is necessary to set the counterpart to serve as the new front body.

3.1.2 Filled chain model

To achieve visual resemblance to realistic cells with amoeboid behavioral patterns, such as a neutrophil, the design must fulfill certain properties. These properties can be observed in real life microscopic footages showcasing neutrophiles moving in blood [10]. First, cells of this variety possess significantly higher deformity attributes than those modeled after a soft-body. Second, any deformity of the cell’s body is shaped by its contents – the majority of which consist of cytoplasm, which can be further simplified as a mildly viscous liquid. The filled chain model has been designed in order to meet the behavioral requirements for this specific, visually distinct type of unicellular organisms.

The physical representation of the filled chain model consists of two components – the chain of circular fixtures and the interior filled with particles (Figure 3.4).

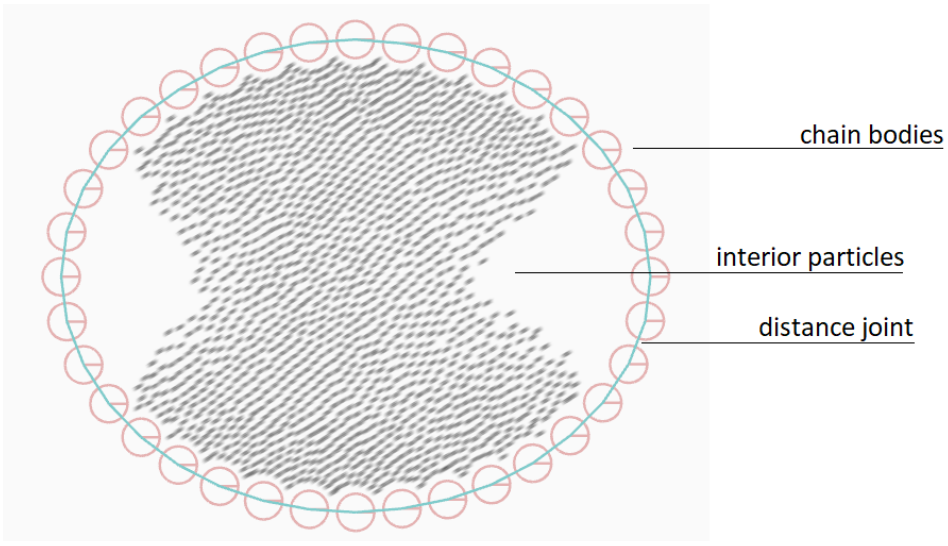


Figure 3.4: Filled chain model structure

The chain structure

The chain serves the purpose of the cell's perimeter which consequently acts as a container for its interior particles. It is comprised of multiple dynamic bodies with circular shapes which are initially distributed in an even manner along a parametrized ellipse. Neighbouring bodies along the ellipse are then connected with ordinary Box2D distance joints to ensure closure. It is advised to create enough bodies to cover the whole perimeter without any large gaps in between them to prevent the interior from leaking in rather unstable cell body configurations.

It is also advised to choose rather small radii for the circular shapes to support deformity properties. In the environment of the implemented simulation, trials have shown that 0.78 (Box2D units) or 2.5 (simulation world units) is the most reasonable value for the circle radius. Note that this value is preferable for any scale of the chain – smaller radii may lead to a lot of instability, while larger values will drastically lower the model‘s deformity.

The interior structure

As proposed above, the interior should resemble a mildly viscous liquid, which by colliding with the perimeter of the cell body, will play a key role in giving shape to it. For this objective, LiquidFun particles provide a fitting solution. Out of all possible particle types supported by this extension, water particles with slightly modified parameters seemed to imitate the behavior of real life cytoplasm the best.

It is expected to fill the majority of the inner area of the closed chain with such particles to prevent unstable cell body configurations. The interior, however, should not be over-dense, as filling it with too many particles will not allow the shape to deform. Finally, the particle radius should have a reasonable value in relation to the length of gaps between the neighboring circular bodies in the chain. If the particles are too small, they will most definitely leak out of the larger gaps at some point in the simulation.

Movement

In order to implement amoeboid movement compatible with this model, it is necessary to feature dynamic partitioning of the cell body into different parts – such as the front, the rear and the left and right wing. We simply assign every body in the chain to one of these parts and then run different logic on otherwise identical chain bodies based on this assignment. Note that every time we update our simulation, we can also reassign the bodies to different parts, thus making the partitioning dynamic.

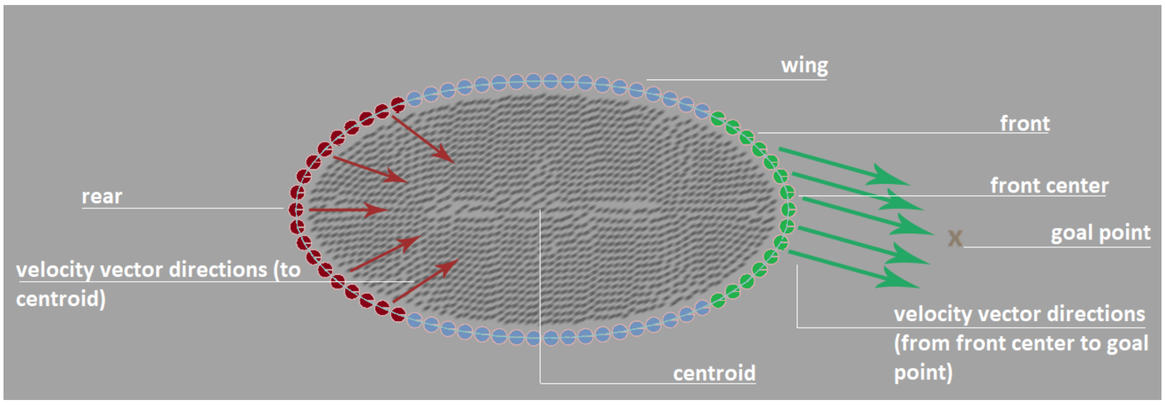

wing
front
rear
front center
goal point
x
velocity vector directions (to centroid)
velocity vector directions (from front center to goal point)
centroid

Figure 3.5: Filled chain model partitioning and movement

Once the cell based on this model receives a movement goal vector (a point in Box2D world towards which the cell should be moving), the closest body on the chain to it becomes the center of the new front. A symmetric section of close neighboring chain bodies in both directions also get assigned to that front. Rear and wings assignment undergo the same process. For movement, only front and rear parts are crucial.

Next, normalized linear velocities are set in all front bodies in the direction of the movement goal vector and in all rear bodies in the direction of the chain's centroid. This approach ensures that the movement is smooth, unidirectional in all parts of the cell and optimally sensitive to input change. This process is depicted in Figure 3.5.

Dynamic scalability

The filled chain model features simple scalability at run time. This includes the extension of both the chain and the interior. The extension of the former is achieved in a straight-forward manner – a new body with identical parameters is added in between two others. In the process, the original distance joint connecting the neighboring bodies is destroyed and replaced by two other joints of the same length and properties, both of which connect the new body with the recently disconnected neighboring bodies. This approach proved to be stable and undemanding of an improvement.

As for the interior extension, a user-assigned amount of particles is added to the existing particle system at the centroid of the chain. Most of the time, the creation of new particles inside the chain will cause temporary overlapping with older particles, to which Box2D responds by applying strong forces upon these overlapping particles. A consequence of that is a visual effect resembling sudden inflation of the cell which clearly signifies enlargement.

However, this behavior contributes to general instability of the simulation, because in some extreme cases, particles with a high linear velocity can push through the distance joints between the circle-shaped bodies of the chain and thus leak. Therefore, it's proper to set an upper bound to linear velocity for all bodies in the chain to prevent such situations.

3.1.3 Absorption models

The filled chain model body described in Section 3.1.2 supports the ability to absorb other simple and complex bodies (specifically soft-bodies). From a biological standpoint, amoeboid cells mostly absorb other objects using phagocytosis [31] – a very complicated process. It can be narrowed down to two visually characteristic properties: first, the cell partially envelops the absorbed object and after that, the cell fully absorbs it.

The goal of the simulated process is to move the absorbed body from outside the chain to its interior as stably as possible. In this case, a stable process ensures that it can only have two conclusions – either the whole absorbed body is surrounded by the chain and thus becomes part of its interior, or it remains outside the chain in its entirety. Emphasis must be put on preventing intermediate states – such as partial absorption. For this reason, absorption of simple single-fixture bodies and complex bodies should be handled differently.

Note that since the simulation is expected to perform with gameplay features, it is appropriate to devise intuitive rules for when the cell can initiate absorption. Such rules define conditions that must be constantly met during a set amount of time. General conditions include:

1. The absorbed object is in direct contact with the absorbing cell.

2. The absorbing cell is moving towards the absorbed object (approximately).

Additional conditions may follow to increase the stability of this process. Violating any condition before reaching the required time will result in a time reset.

Once the cell successfully initiates the absorption process, the gateway into its interior is opened by disabling collision between all of its current front bodies and the absorbed object. As mentioned before, the next steps will use different approaches depending on what kind of body is being absorbed.

Absorption of single-fixture bodies

Single-fixture bodies consist of only one, simple fixture. In the implemented simulation, it is always circle-shaped. Provided it is properly scaled to fit the absorbing cell's interior, this kind of absorption is expected to occur more frequently. Thus, it is necessary to consider cases in which multiple objects will be absorbed simultaneously.

The aforementioned general conditions are sufficient to start this type of absorption. Once they are met for an individual object, the object becomes absorption-ready and the collision is disabled so that it can enter the cell's interior. At the moment any object acquires this status, a time counter will be initiated. At the end of its time limit, the object will lose its absorption-readiness, collisions will be restored and the process will terminate with only two possible outcomes:

1. The object is completely engulfed by the cell's chain bodies.
2. The object remains entirely outside the cell‘s chain bodies.

Due to the simplicity of the fixture, any intermediate states are prevented. For example, if it happens that the collision is restored right at the moment when the object is overlapping with a chain body, Box2D's inner logic will simply push that object either inside or outside the chain. Thus, unstable partial absorption is impossible in this case, as desired in our simulation.

The simulation should be able to determine which of the two outcomes took place. From this, the necessity of an algorithm emerges. This algorithm should be capable of providing an unambiguous answer to whether the object being absorbed is in the interior or exterior of the cell.

The ray-casting algorithm [32] is a fitting solution.

Ray-casting algorithm. A set of polygon-defining vertices and the tested point are its input. In terms of the filled chain model body, the coordinates of these vertices correspond to the centers of individual chain bodies. The tested point matches the center of the absorbed body.

A horizontal ray is cast in one direction from the tested point. The number of its intersections with the polygon edges is then counted. If this number is odd, the tested point belongs to the interior of the polygon. If it is even, it is part of the exterior.

To improve efficiency, this algorithm only tests objects that are in the state of absorption-readiness (i.e. objects that have met the aforementioned conditions). Note that such objects maintain their status for a limited amount of time. As a result, the number of times the algorithm should be executed in one frame is constantly reset and does not exceed extreme values.

Absorption of soft-bodies

A soft-body's structure is complex and consists of multiple joint bodies. Furthermore, they are generally expected to be of a larger scale. Both of these assumptions could prove very problematic in terms of the absorption process stability.

An additional precondition must be met in order for the process to start - the filled chain model-based cell must now have at least several collision contacts with the soft-body. Once the process is initiated, the absorbing cell temporarily interrupts its movement and does not react to user input. This significantly limits the span of possible shapes the cell can appear in at that moment. Many shapes may unpredictably cause problems during this stage, so it is suitable to prevent them.

The object of absorption loses its independent mobility and moves towards the centroid of the absorbing cell for a limited amount of time. The time is set so that the soft-body is guaranteed to reach its goal before the limit. In the duration of this process, the soft-body also continuously shrinks to stably fit into the interior. Once the limit expires, the soft-body is then marked as absorbed, collision is restored and the absorbing cell regains its movement.

3.1.4 Models for other microscopic objects

Platelet

The proposed models for unicellular organisms in the simulation require a fitting environment to interact with. To make the simulation more diverse, a suitable object acting as an obstacle can be designed.

So far, the simulation has been shaped in accordance to real life references from the microscopic life. Many of these references originate from microscopic footages of mammalian blood. Besides neutrophils, an apparent presence of numerous objects called blood platelets (also known as thrombocytes) can be observed in Figure 3.6. It is noticeable that neutrophils often move through the gaps between platelets when chasing other organisms – a fitting potential feature for our simulation. Furthermore, platelets seem to have very similar properties to those of a simulated soft-body. It stands to reason that a blood platelet makes for a suitable reference when designing the form of an obstacle object.

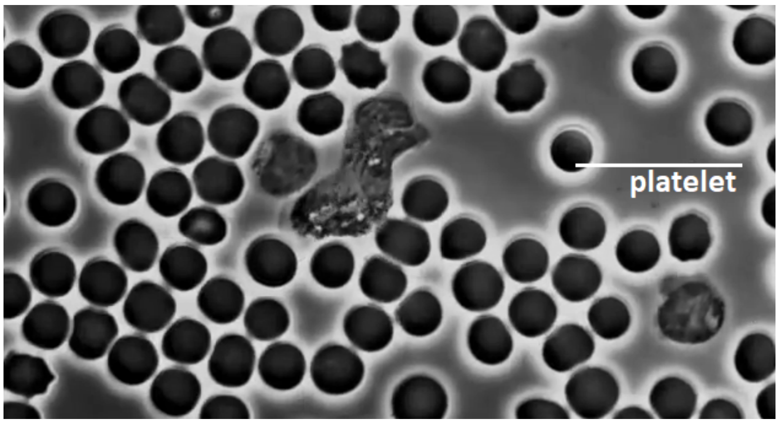


Figure 3.6: Platelets in real life [2]

In the simulation, platelets are expected to constitute the most numerous category of microscopic objects. Thus, emphasis must be put on performance, which is dependent on the total number of simulated bodies. As mentioned above, the only criterion for visual resemblance to the reference material is to include soft-body attributes. However, a double-layered soft-body is a model that is very demanding of a large number of circular bodies that constitute it. Reducing it to a single layer cuts the number in half, but lowers its overall stability – an important feat, considering its function as an obstacle.

A modified soft-body model specifically optimized for platelet simulation has been devised to address these issues (Figure 3.7). It consists of a single layer of satellites, the kernel and two new additions in the form of the so-called interlayer and anchor point. The interlayer serves as a performance-friendly alternative to the second layer in soft-bodies, providing comparable stability at the sole cost of lesser deformity.

It is represented by a dynamic rigid circle-shaped body. The center of it is aligned with the center of the kernel. Overlapping is ignored due to collision disablement between both bodies. The design shows best results when the interlayer's radius is large enough for it to almost reach to the satellite bodies.

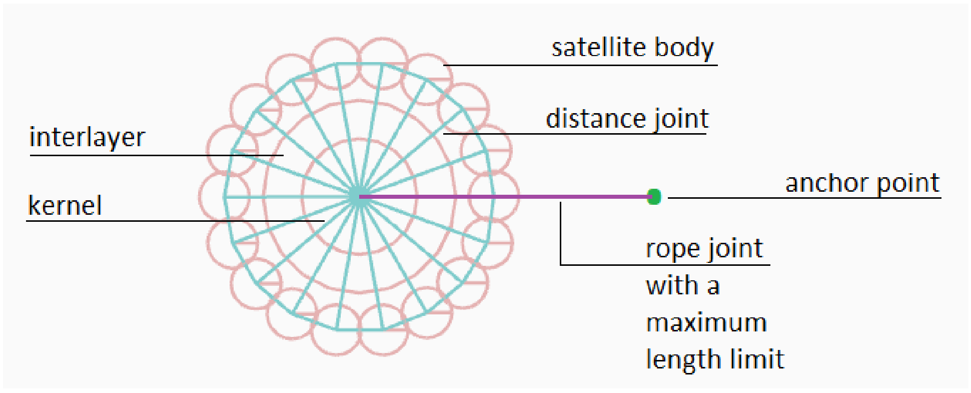


Figure 3.7: Optimal soft-body structure for platelets

The soft-body‘s kernel is bound to a non-colliding static point called the anchor point. The purpose of this concept is to set an upper bound to the distance between a platelet and a static point in the Box2D world. Not only does this limit the platelet‘s free movement in the environment, it also allows unicellular organisms to push through the platelets – a realistic feature resembling the reference footage.

In Box2D, we can realize this type of connection by using a rope joint. Unlike distance joints, rope joints support upper bounds on their length. With further customization, rope joints in the simulation have been set to have a tendency to contract to minimal length. This means that platelets that have been pushed from their initial position will eventually return to it.

Agar

Besides other unicellular organisms serving as a source of absorbable nourishment for the controlled cell, it is suitable to design a more common and numerous unit meant for casual absorption. In the referenced game Agar.io, such role was assigned to small colored circles that were densely arranged on the map. These objects have been inspired by agar [33], a real life source of nutrients in the microscopic world. Analogically, simple small circle-shaped rigid bodies can be placed to serve the same purpose in a physics-based simulation, such as the subject of this work.

Considering the fact that agar objects are expected to be absorbed very frequently and in larger quantities, the simplicity of the representation allows the use of a more stable absorption process, such as the one mentioned in Section 3.1.3.

3.2 Rendering methods

The models of various microscopic objects proposed in Section 3.1 only include their physical representation. The following section describes the applied methods of binding visual representation to its physical counterpart in Box2D, as well as secondary visual elements contributing to the broad vivacity of the simulation view.

All used methods will utilize basic functionality from the LibGDX library.

3.2.1 Rendering soft-bodies

All simulation objects modeled after the soft-body pattern use a two-dimensional LibGDX mesh [34] as its main component for rendering. It is configured to hold four-dimensional vertices in the following format – the first pair describes the horizontal and vertical coordinates of the vertex on screen, while the second pair refers to texture coordinates. Note that every vertex in the mesh corresponds to exactly one physical body that is part of the outermost layer in the soft-body.

The vertex coordinates are in alignment with the centers of these physical bodies. However, if it happens that the circular shapes of these bodies possess a relatively large radius, visual imperfections in form of significant gaps will become evident during collisions. In Figure 3.8, it is visible that the rendering would only reach the red points inside the circles if not adjusted. It is therefore useful to extend the drawing surface all the way to the yellow points to cover the area beyond the circle centers.

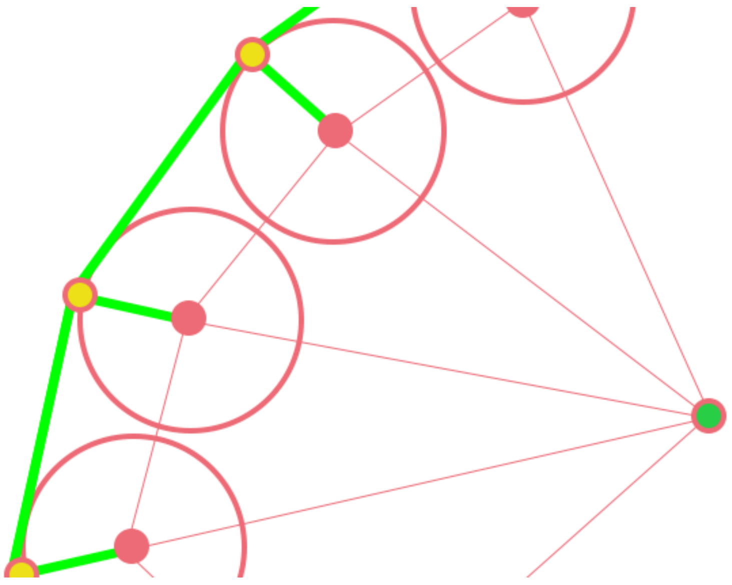


Figure 3.8: Extended drawing surface covering

For each circle-shaped body, an adjustment of this sort can be achieved by setting a local center point inside the soft-body. In Figure 3.8, the green point acts as a local center. After that, the direction from that local center to the center of the target circle is calculated. Finally, a vector of that direction is scaled to match the radius of the circle shape in length and added to the body's positional vector.

Using the texture coordinates, certain points in the texture image can be bound to physical bodies. If properly mapped, the drawn texture will automatically transform in response to the detected deformations in the physical representation. Note that these coordinates will not change throughout the simulation, therefore no additional managment is necessary besides initialization. When designing the texture, it should be taken into consideration that the soft-bodies used in this simulation are either circle- or ellipse-shaped. In order to simplify the initial mapping, square textures with an inscribed circle are recommended. That way, simple trigonometry is sufficient to bind circle-shaped bodies in the outermost layer of the soft-body to the same amount of evenly distributed points along the inscribed circle. The center of the texture is then mapped onto the position of the soft-body's kernel.

After setting up the mesh vertices, indices referring to them will be used to render the contents of the mesh in a triangular layout. Each triangle will be described with an index triplet corresponding to three specific vertices in the mesh. Since the soft-body's layout is triangular by definition, these indices will remain static throughout the whole simulation. Every triangle will feature one vertex bound to the soft-body's kernel, as well as two others bound to neighboring circle bodies in the soft-body‘s outermost layer. The kernel of the soft-body is partially mobile within its boundaries, therefore it is suitable to equip it with a separate visual representation. Using a properly scaled sprite based on a texture for that purpose is sufficient to conclude the graphics of a soft-body.

3.2.2 Rendering filled chain model bodies

Similarly to soft-bodies, filled chain model bodies also use a two dimensional LibGDX mesh. However, the first difference can be observed in its vertex format. In this case, the vertices are simply two-dimensional. They only hold their horizontal and vertical coordinates on the screen which are managed the same way as in the soft-body. In a filled chain model body, these coordinates adjust to the positions of individual chain bodies.

The absence of texture-mapping relates to the fact that the mesh indices are expected to change very frequently, as opposed to being immutable like in the soft-body mesh. Due to a generally higher deformity rate, the filled chain model body lacks an inner point, such as the soft-body kernel, that serves as a common vertex for all triangles in the layout. Thus, the complex shape cannot follow a predetermined triangular layout. A triangulation algorithm must be applied instead. LibGDX offers an implementation of the Ear clipping algorithm [35] which can be used to divide the complex shape area into triangles and output respective index triplets (Figure 3.9). If texture-mapping was applied in such layouts, it would not produce smooth transformations and only come across as a visual shortcoming. Instead, the triangulated area is simply filled with one color.

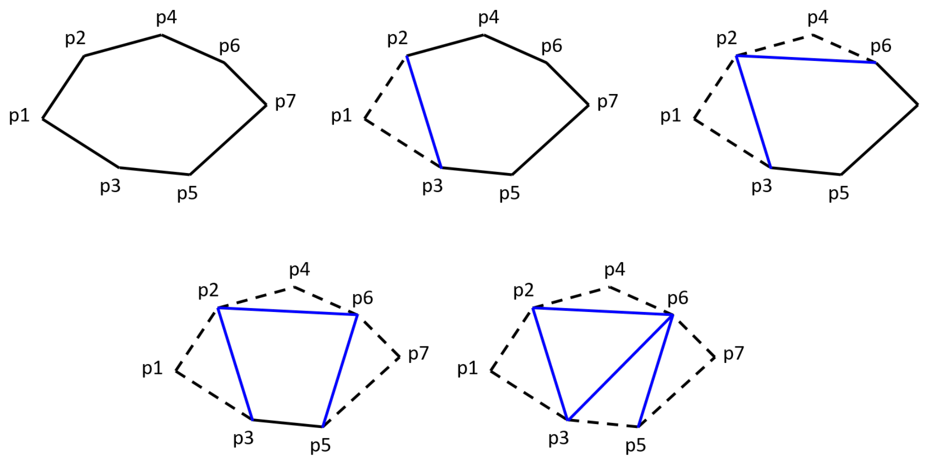


Figure 3.9: Triangulation process of a polygon [3]

Atop that color, inner particles will be used to create visual complexion. For the purposes of this simulation, the default Color particle renderer, which is included in the LiquidFun extension, will suffice. It is capable of drawing every individual particle as a filled circle of respective radius and color with reasonable performance.

Since the inner particles represent real life cytoplasm in the simulation, their visual representation should resemble a continuous fluid. Such result can be achieved by applying a simple Gaussian blur to the rendered particles.

3.2.3 Rendering background

Although it seems that the view under a real-life microscope is two-dimensional, microscopic life is certainly three-dimensional. Therefore, it is suitable to add another layer of objects to simulate some degree of depth in the simulated view. When rendering the objects in the background, it is needless to include their physical representation – their purpose is strictly visual. They are simply represented by a two-dimensional point in the Box2D world and a sprite reference. The sprite's position is then aligned with that point every time it changes its coordinates. In the implemented simulation, this position changes randomly. The resulting rendering can be seen in Figure 3.10.

Figure 3.10: Rendered background

3.3 Application

The physical models and their visualization methods proposed in the previous sections will serve as tools to create an application showcasing the resulting simulation.

Although the application should be runnable on multiple platforms due to the advantages of the LibGDX framework, note that its development is mainly aimed at desktop platforms. The application does not put emphasis on optimization and mainly prioritizes visual appeal over potential efficiency gain. As a result, the application is designed to run stably at 60 frames per second on most desktop platforms, but further improvements in performance might be necessary for platforms such as mobile.

The application consists of two closely related components – the simulation and game mode. The former serves as a sandbox mode, fulfilling the purpose of demonstrating all features included in the proposed models. It can be seen as a pure simulated view under a microscope. The latter is an extension of the former. It adds rules, gameplay and general meaning to the simulation by utilizing its features.

3.3.1 Simulation mode

To set the Box2D-based simulation in motion, its world must be filled with simulated objects. These objects have the shape of the proposed physical models and thus support their functionality. This stage establishes relations between these individual features and adds logic to when they are triggered and what consequences they cause. A number of supporting processes are defined.

Simulation objects

The simulation's environment is filled with various objects (Figure 3.11) directly derived from the proposed physical models. These include living unicellular organisms, which are expected to periodically receive movement input, and static auxiliary structures. The input is either directly received from the user, or generated by a trivial random artificial intelligence.

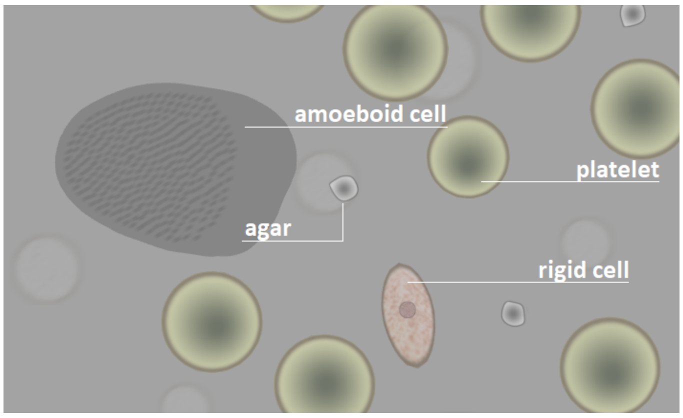


Figure 3.11: Supported simulation objects

Rigid cells. Represented by the soft-body model, rigid cells constitute one of the living organism varieties inhabiting the simulation environment. These cells serve the purpose of adding vivacity to the simulation, as well as potential nourishment for amoeboid cells.

Amoeboid cells. Filled chain model bodies are used to simulate cells of the amoeboid variety. Due to a higher rate of deformity, elaborate functionality and visually distinct interactions with its environment, it is the most suitable simulation object for user control. Relating to the supported features of the filled chain model body, an amoeboid cell can absorb agar and rigid cells, move around in the environment and increase in scale.

Obstacles. To fully demonstrate the deformity features of the filled chain model body, numerous soft-body obstacles are placed within the simulation. They are represented by the implemented platelet model.

Agar. Agar is another non-living object contained in the simulation. Agar serves as a common unit of absorption and consumption for amoeboid cells.

Walls. Rectangular boundaries demarcate the active part of the simulation and thus prevent any object from leaving them.

Object consumption

Once an amoeboid cell absorbs a foreign object, it remains in its interior. Since these objects may accumulate over time, it is suitable to devise a process that would eliminate such objects. By making this process manual, it may simultaneously serve as a potential game mechanic.

The process shrinks the object until a certain scale is reached and consequentially, the object is fully deallocated from memory. The interior extension described in Section 3.1.2 is bound to the end of this process. As a result, the only possible way of enlarging an amoeboid cell is by consuming objects, not just absorbing them.

Absorption management

The simulation considers the potential instability that might occur during the absorption of complex bodies and is therefore equipped with a simple system that puts dynamic constraints on what rigid cells can be absorbed at a certain time.

These constraints mostly refer to the sizes of both actors in the absorption process. Size of a complex body is related to the area defined by its outline vertices. Since the area should remain constant unless the complex body is dynamically scaled (such as the amoeboid cell after consuming an object), deformations will be ignored in the resulting area calculations.

Rigid cell area. In terms of our simulation, the area of rigid cells corresponds to the area of the initial ellipse shape. Only minor deformations occur in soft-body based objects. Therefore, this constant value describes its size quite accurately at all times.

Amoeboid cell area. Area of an amoeboid cell is defined as the sum of its inner particles' areas. While mathematically inaccurate, it is sufficient for the purposes of this simulation. As the scale of an amoeboid cell grows, the chain of its body is extended and its interior is filled with more particles at the same time. It is more difficult to make adjustments to the area based on the extension of the cell‘s perimeter alone, therefore its filling is applied in calculations instead.

A ratio is established between the area of the amoeboid cell and the rigid cell. The simulation allows the player cell to absorb only those rigid cells which meet this ratio. Even though rigid cells are scaled down when they enter the player cell, they still take up much more space than agar units. Thus a limit is set so that the player cell can hold at most one absorbed rigid cell in its interior until it is consumed.

Demonstration environment

The implemented application includes a manually designed environment (Figure 3.12) for the purpose of demonstrating its entire functionality. It features all of the supported objects with different parameters that determine their size, behavior and other properties.

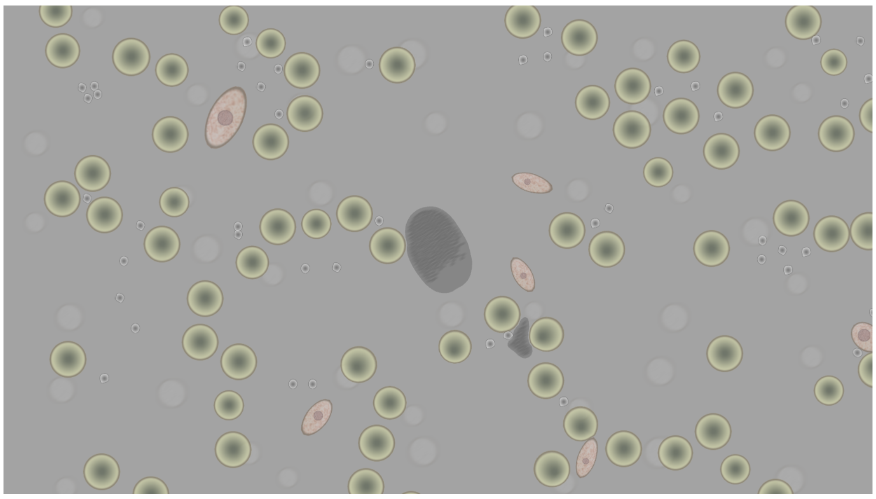


Figure 3.12: Demonstration environment

3.3.2 Game mode

A prototype of a game has been implemented as part of this thesis to show the utilization potential of the proposed solution. Although the prototype is single player, it adapts many gameplay mechanics from the critically acclaimed Agar.io [4] game.

Rules

The user plays as an amoeboid cell that has 90 seconds to increase its score. Score is increased by consuming an absorbable object - agar or rigid cells. Agar can be absorbed and consumed in any instance, but its consumption only adds a small amount of points. Since rigid cells require the player cell to reach a certain size in order to allow their absorption, the following consumption earns more points.

Once an object is absorbed, the player can devise a strategy on when to consume it. Consumption is a manually initiated process that significantly slows down the cell for the entirety of its duration.

As previously mentioned, the simulation associates the amoeboid cell enlargement with the successful termination of the consumption process. Besides the acquired points, the player cell will also become larger and thus capable of absorbing rigid cells of greater scale.

HUD and other visual elements

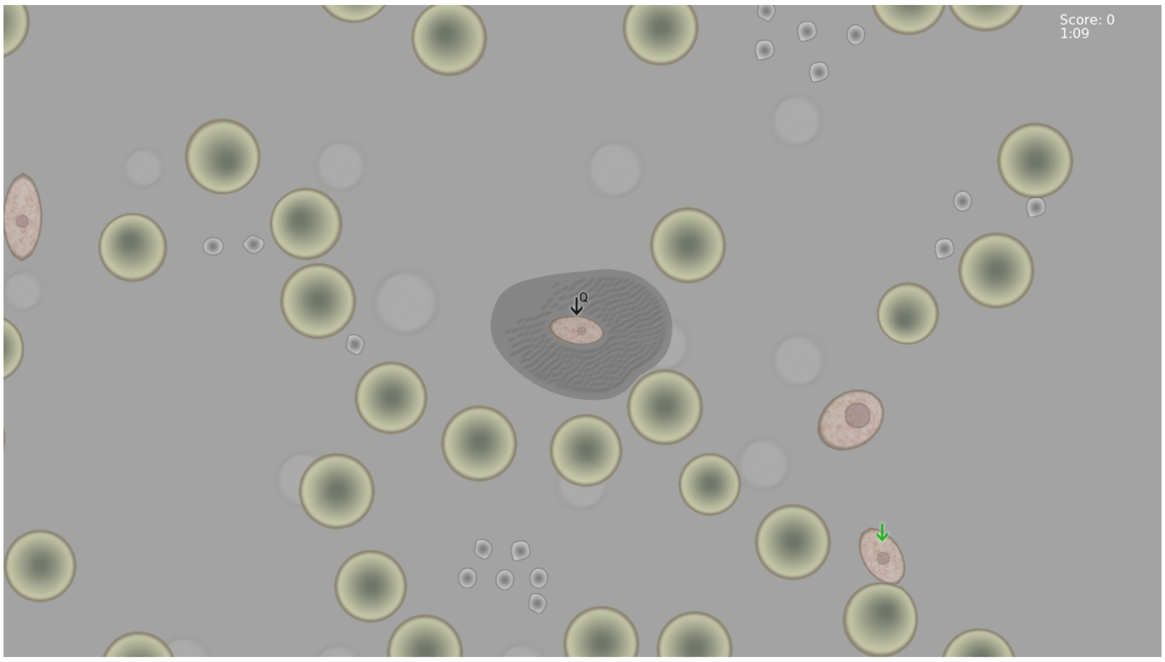


Figure 3.13: Game mode screenshot

HUD (head-up display) refers to a game's user interface. It contains static visual elements that display the game's status variables. These include the score view and the time limit.

Other visual elements are rendered by the game mode on top of the simulation (Figure 3.13) to futher assist players. The list contains:

1. Green arrow. It points to rigid cells that are ready to be absorbed by the player cell.
2. Score gain view. A number will show the amount of points the player has received after consumption.
3. Consumption marker. It will point to absorbed objects that will be consumed upon holding the displayed key.

Chapter 4

Results

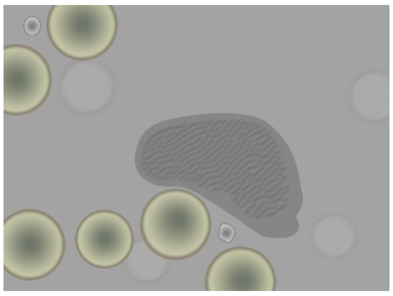
The following chapter evaluates the resulting simulation in terms of its visual fidelity, performance and stability. It lists its advantages and disadvantages over rivaling simulations, known imperfections and ways of usage in future work.

4.1 Comparison

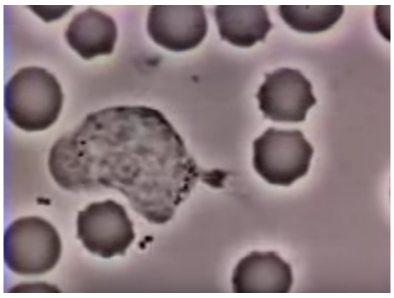
4.1.1 Visual comparison

General appearance

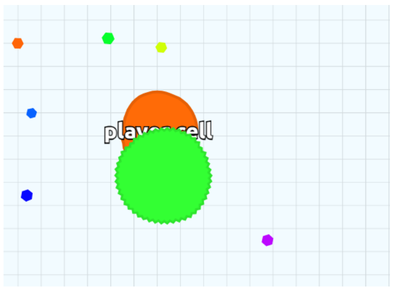
The amoeboid cell implemented in this work clearly surpasses its competition in terms of visual fidelity, as seen in Figure 4.1. Compared to the real life example, however, it still lacks complexion in its interior. A closer resemblance can be achieved by making the individual interior particles have different sizes and colors.



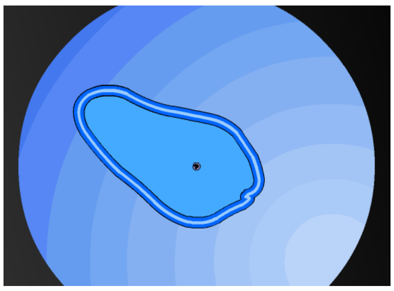
(a) : Thesis result



(b) : Real life [10]



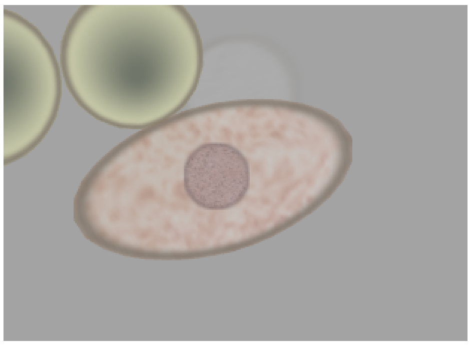
(c) : Agar.io [4]



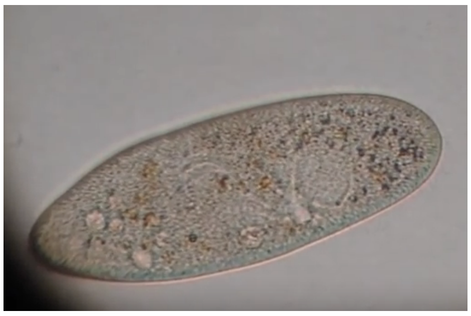
(d) : CellCraft [14]

Figure 4.1: General appearance comparison of amoeboid cells

Rigid cells, which were inspired by real life paramecium cells, do not have a counterpart in competition. Figure 4.2 shows that although the shape bears sufficient resemblance to a paramecium cell, the complexion is different in both style and structure.



(a) : Thesis result

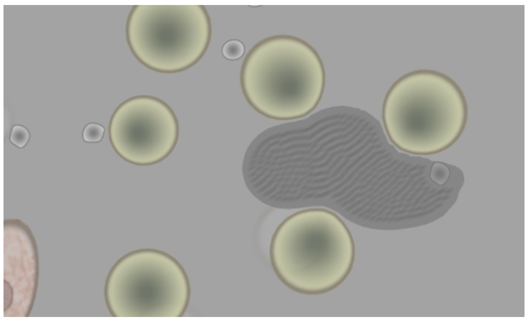


(b) : Real life [36]

Figure 4.2: General appearance comparison of paramecium-inspired cells

Deformative qualities

Deformative properties are best seen in amoeboid cells. Similar configurations are shown in Figure 4.3, as both cells are shaped by their surroundings.


(a) : Thesis result

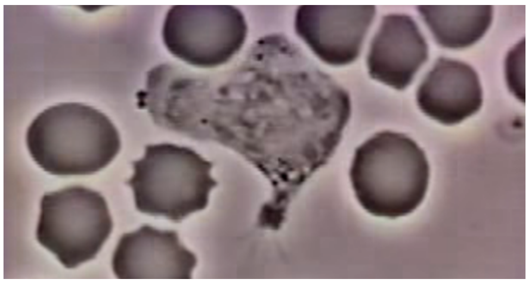
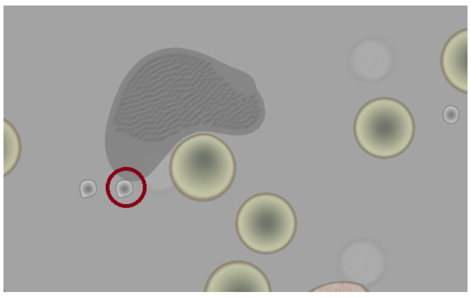
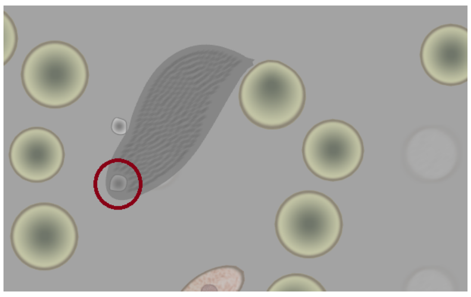

(b) : Real life [10]

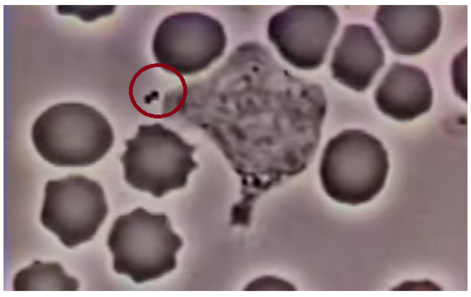
Figure 4.3: Deformative qualities comparison of amoeboid cells

Absorption

All existing non-scientific solutions mentioned in Section 2.2.1 only support absorption on a logical level. It means that once a collision is detected with an absorbable object, it instantly disappears instead of actually entering the cell. The thesis result, however, adds physical level to the absorption process. The absorbed object can become part of its interior. This contributes to realism, as seen in Figure 4.4.


(a) : Thesis result (before)


(b) : Thesis result (after)


(c) : Real life (before)

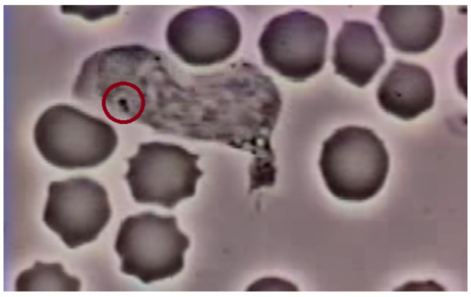
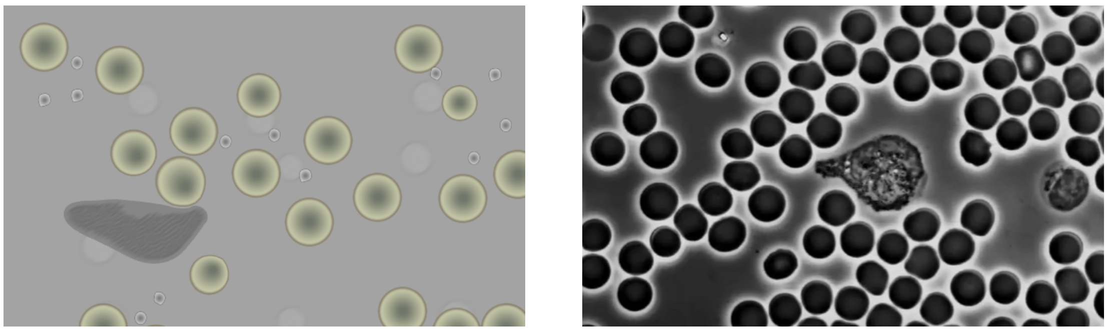

(d) : Real life (after)

Figure 4.4: Comparison of absorption in amoeboid cells

Environment

Platelets constitute the main component of the simulated environment. In terms of the objects themselves, a rather high degree of visual fidelity has been reached (Figure 4.5). Further improvements can be made by slightly distorting the shape of the simulated platelets. Differences in the quantity of platelets are caused by performance issues that will be mentioned in Section 4.2.



(a) : Thesis result **(b) :** Real life [2]

Figure 4.5: Environment comparison

4.1.2 Features comparison

Game mechanics aside, the implemented simulation supports all but one critical feature found in Agar.io [4]. The player cell can move, absorb other objects and get larger, all with a greater degree of visual fidelity. In Agar.io, however, the player cell can also be divided into multiple smaller ones, referring to real life cell reproduction. It is possible to implement a likewise feature in the Box2D environment by fittingly breaking and reattaching distance joints. This feature goes beyond the scope of this thesis, however.

4.2 Limitations

Physics-based simulations tend to be generally more advanced in terms of visual fidelity, but many simulations choose to deliberately avoid the inclusion of physics to save on performance and increase stability. The limitations of the simulation implemented within the scope of this thesis relate to general problems of physics engines - in this case, specifically Box2D.

4.2.1 Performance

The application was run on the following hardware components when observing its performance:

- GPU: NVIDIA GeForce GT 630M with 2 GB Dedicated VRAM
- CPU: Intel Core i5-3210M

Physics

The application predominantly uses the CPU to simulate dynamic bodies contained in the Box2D world. It follows that the number of dynamic bodies in the simulation constitutes the decisive criterion for performance.

Most dynamic bodies belong to the numerous platelets inhabiting the simulation environment. In the current revision, platelets are based on soft-bodies. With parameters that grant a decent degree of visual fidelity, each platelet is expected to add 20 to 30 dynamic bodies to the total number. Without platelets, the environment leaves an empty impression. Therefore, the scale of the simulation map can be closely associated with the number of platelets in it.

Simulated unicellular microorganisms tend to increase the total count of dynamic bodies as well. In fact, the double layered soft-body featured in rigid cells has the highest number of dynamic bodies per simulation object. However, these objects are not expected to be densely placed within the simulation.

The following benchmarks depicted in Table 4.1 have been observed when running the simulation without rendering:

Number of dynamic bodies	Number of platelets	Frames per second
3000	90	60
4000	120	60
5000	150	50
6500	200	30

Table 4.1: Benchmarks on simulation performance (physics only)

The first line in the table approximately relates to the scale of the simulation map used in the application's demonstration. Issues started to appear when Box2D simulated over 4000 dynamic bodies, roughly corresponding to a 50% increase in map scale.

Surpassing the 6500 dynamic bodies mark, frames per second started to drop drastically under the value of 30 - leaving the simulation in an unplayable state.

If map scale is of essence, platelets can be optimized to be represented by a single dynamic body instead of a complex soft-body. By significantly cutting the number of bodies contained in one platelet, the simulation can achieve massive scales while maintaining stable performance.

As a last resort solution, simulation objects that are not visible to the user can be set to sleep mode. This way, the Box2D internal logic will temporarily not consider them in its collision and velocity calculations. Once they enter the vicinity of the camera again, they are awoken. Note that such solution brings many other limitations to the simulation and can be used only in case of a single player game.

Rendering

LibGDX offers a variety of automatic optimizations to render calls. A lot of room is left for further manual improvements, such as limiting render calls by disabling managment of objects outside the camera vicinity. For the purpose of our simulation, however, automatic features suffice in providing reasonable performance. Rendering issues are strongly overshadowed by the difficulty of physics-based computations.

4.2.2 Stability

When building complex custom structures in Box2D, it is sometimes a difficult task to predict their behavior during interactions with other objects. As a rule, the more primitive an object's structure is, the more stable it acts in the simulation.

Primitive objects are not sufficient for this work's purposes, however. As a downside, the simulation struggles with virtually unavoidable potential instability. Although most of its instances have been suppressed, some of them remain unresolved. Note that these imperfections are generally unlikely to happen and are mostly caused by unfit controlling schemes. In the entirety of the testing process for this application, not a single program-crashing error was detected.

Two most apparent problems will be described. Note that other potential problems are not excluded, numerous trials might unveil bugs that were previously unheard of.

Improper absorption

This problem occurs when the user-controlled cell continuously pushes a rigid cell. Normally, rigid cells are absorbed when pushed by an amoeboid cell. Exceptions might arise if the user is improperly controlling the cell, the rigid cell cannot be absorbed or the player cell already has an unconsumed rigid cell in its interior.

When the front part of the chain in the amoeboid cell constantly pushes the outermost layer of satellites in the rigid cell, the distance joints between two chain bodies eventually extend and allow the rigid cell to enter the player's interior (Figure 4.6). Note that this action is not handled by the absorption process. Therefore, the improperly absorbed rigid cell still maintains motion and cannot be consumed, even if physically inside the amoeboid cell.

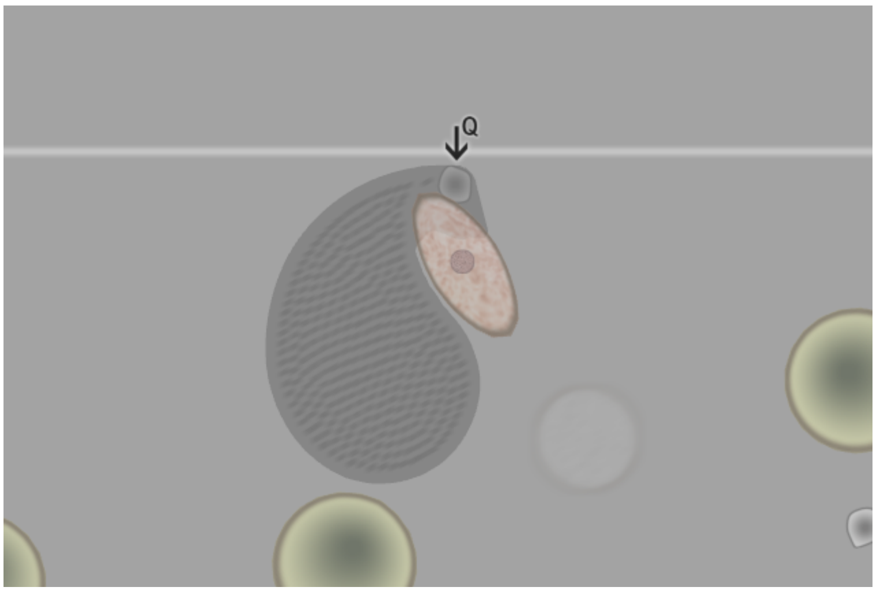


Figure 4.6: Improper absorption bug

A possible solution might be a custom implemented distance joint that will respect a maximum length limit.

Amoeboid cell leakage

Although the exact causes of this problem are left unknown, in some very rare occurrences possibly related to amoeboid cell enlargement, inner particles may be discharged under pressure (Figure 4.7). Unlike the former problem, this only causes a visual inconvenience and should not affect the gameplay in any way.

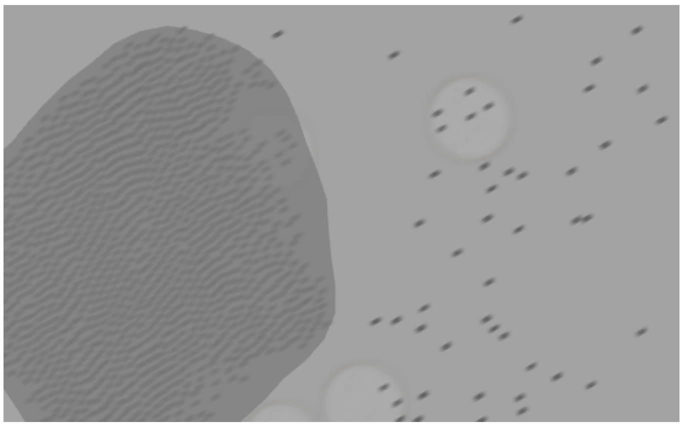


Figure 4.7: Amoeboid cell leakage bug

A possible solution is to limit the maximum velocity of the particles so that they cannot penetrate though the distance joint between the individual chain bodies of the amoeboid cell's filled chain body.

4.3 Future work

The project leaves a lot of space for further improvement. In addition to fixing some of the limitations mentioned in Section 4.2, the implemented prototype of the game can be used to develop a fully-fledged game, possibly with multiplayer support.

By optimizing the code and simplifying some features, the simulation can also be ported to mobile devices and web browsers.

By adding some educational commentary, the simulation can also be used as an introductory tutorial lesson to cellular biology in primary and secondary schools.

Chapter 5

Conclusion

This thesis provided a brief introduction to the topic of microscopic simulations, their niche and potential uses in educational, recreational and scientific fields. By analyzing existing solutions with imperfect visual fidelity, innovative methods were devised to surpass its competition. In contrast to rivaling solutions, the chosen methods strongly relied on two-dimensional physics simulation. Several physical models were designed to represent various objects from the microscopic world. Special emphasis was put on shaping the form and functionality of simulated unicellular microorganisms, especially those of the amoeboid variety.

The dominant output of this work is the implementation of a desktop application that utilizes the designed theoretical models. A broad variety of capabilities were demonstrated within the implemented simulation, one of which is the option to be used in video games.

Although the proposed solution has successfully reached its goal of achieving the highest degree of visual fidelity amongst existing non-scientific simulations of this sort, an honest assessment of its disadvantages was given.

Bibliography

[1] Umberto Salvagnin. Onion cells 2. https://www.flickr.com/photos/kaibara/3839720754/, 2009. Accessed: 2018 May 18.

[2] Kim Ulvberget. White blood cells and bacteria. https://www.youtube.com/watch?v=DXANDu083J4, 2014. [online video] Accessed: 2018 March 20.

[3] Amrita Vishwa Vidyapeetham. Polygon triangulation. https://www.amrita.ac.in/swaminathanj/cg/PolygonTriangulation.html. Accessed: 2018 May 18.

[4] Valadares Matheus. Agar.io. http://agar.io, 2015. Accessed: 2018 May 16.

[5] Google trends. https://trends.google.com/trends/topcharts#vm=cat&geo&date=2015&cid, 2015. Accessed: 2018 February 20.

[6] John Fingas. Agar.io brings massively multiplayer games to the petri dish. https://www.engadget.com/2015/06/01/agar-io/, 2015. Accessed: 2018 February 19.

[7] N.A. Campbell, B. Williamson, and R.J. Heyden. *Biology: Exploring Life*. Pearson Prentice Hall, 2006.

[8] Kenneth Todar. Structure and function of bacterial cells. http://textbookofbacteriology.net/structure_2.html, 2008-2012. Accessed: 2018 May 18.

[9] Blanka Škrabálová. Prvoci [protozoa]. http://www.mikro.jaknahmyz.cz/prvoci. Accessed: 2018 May 18.

[10] David Rogers. Crawling neutrophil chasing a bacterium. https://www.youtube.com/watch?v=I_xh-bkiv_c, 1950s. [online video] Accessed: 2018 February 2.

[11] Keanna Burns. Otherwordly amoebas. *Future Science Leaders*, 2014.

[12] Maxis Games. Spore. http://www.spore.com/, 2008. Accessed: 2018 May 18.

[13] Hemisphere Games. Osmos. https://www.osmos-game.com/, 2009. Accessed: 2018 May 18.

[14] Lars A. Doucet, Chris Gianelloni, Anthony Pecorella, and Hibiki Haruto. Cellcraft. https://www.biomanbio.com/GamesandLabs/Cellgames/cellcraft.html, https://github.com/larsiusprime/CellGame--Open-Source-fork-of--CellCraft--/, 2014. Accessed: 2018 May 18.

[15] Marc Herant, Volkmar Heinrich, and Micah Dembo. Mechanics of neutrophil phagocytosis: experiments and quantitative models. *Journal of cell science*, 119(9):1903–1913, 2006.

[16] Shin I Nishimura, Masahiro Ueda, and Masaki Sasai. Non-brownian dynamics and strategy of amoeboid cell locomotion. *Physical Review E*, 85(4):041909, 2012.

[17] Erin Catto. Box2d. http://box2d.org/. Accessed: 2018 May 18.

[18] Unity Technologies. Unity3d. http://unity3d.com/. Accessed: 2018 May 18.

[19] Youzu Stars. LÖVE. https://love2d.org/. Accessed: 2018 May 18.

[20] YoYo Games. Gamemaker: Studio. https://www.yoyogames.com/gamemaker/. Accessed: 2018 May 18.

[21] Cocos2d. www.cocos2d-x.org/. Accessed: 2018 May 18.

[22] Badlogic Games. Libgdx. https://libgdx.badlogicgames.com/. Accessed: 2018 May 18.

[23] Erin Catto. Box2d v2.3.0 User Manual. http://box2d.org/manual.pdf, 2013. Accessed: 2018 May 10.

[24] jBox2D. http://www.jbox2d.org/. Accessed: 2018 May 18.

[25] webfanatic. Constant volume joint for boxweb2d - get jiggly with it. https://www.youtube.com/watch?v=TmB6KAxzues, 2013. [online video] Accessed: 2018 April 20.

[26] Chipmunk2d. https://chipmunk-physics.net/. Accessed: 2018 May 18.

[27] Google. Liquidfun. http://google.github.io/liquidfun/, 2013. Accessed: 2018 May 18.

[28] Finnstr Productions. Libgdx LiquidFun extension. https://github.com/finnstr/gdx-liquidfun-extension, 2014. Accessed: 2018 May 18.

[29] Khronos Group. OpenGL. `https://www.opengl.org/`. Accessed: 2018 May 18.

[30] cathlee35. Paramecium. `https://www.youtube.com/watch?v=fmwN_mD7TvY`, 2007. [online video] Accessed: 2018 May 14.

[31] Wikipedia. Phagocytosis. `https://en.wikipedia.org/wiki/Phagocytosis`. Accessed: 2018 May 17.

[32] Rosetta Code. Ray-casting Algorithm. `https://rosettacode.org/wiki/Ray-casting_algorithm`. Accessed: 2018 April 25.

[33] Wikipedia. Agar. `https://en.wikipedia.org/wiki/Agar`. Accessed: 2018 May 17.

[34] Matt DesLauriers. LibGDX Meshes. `https://github.com/mattdesl/lwjgl-basics/wiki/LibGDX-Meshes`, 2014. Accessed: 2018 April 28.

[35] David Eberly. Triangulation by ear clipping. *Geometric Tools*, 2008.

[36] PondWaterWorld 2.0. Paramecium discharging trichocysts. `https://www.youtube.com/watch?v=kHOOtPhf6EM`, 2012. [online video] Accessed: 2018 March 21.

Appendix A

CD contents

- **sencukri_bthesis.pdf.** Thesis text in .pdf format.
- **simulation.jar (located in *app* folder).** An executable program that contains the application showcasing the implemented simulation. Java Runtime Environment is required to run this program. Supported by Windows and Linux.
- **Asset files (located in *app* folder).** Various asset files required by the application (shaders, images, fonts). They need to be in the same directory as simulation.jar.
- **Project source codes (located in *source* folder).** A Gradle-based project and its source codes. Implemented logic can be found in UnicellularOrganismSimulation\core\src.

Appendix B

User manual

B.1 Modes

The application allows the player to choose from two modes.

- Simulation mode: A free mode without any game elements or rules. No hints enabled. Suitable for debugging and demonstrative purposes.
- Game mode: This mode includes gameplay features.

B.2 Game mode rules

You have 90 seconds to increase your score. Score can only be increased by absorbing and consequentially consuming agar or rigid cells (refer to Figure 3.11 for their appearance). Agar can be absorbed at any time, but to absorb certain rigid cells, your cell must become larger. A green arrow will appear above the rigid cell if it is ready to be absorbed.

Your cell becomes larger once an absorbed object is consumed. Simply absorbing an object is not enough - it will remain inside your cell until you completely consume it. Note that the consumption process slows down the cell, so pick a suitable time for it. Since the newly added inner particles are differently colored than the initial onces, you can see how much your cell has grown since the beginning.

Note that only one rigid cell can be inside the player cell at one time. If you have one inside, **you will not be able to consume other rigid cells, even if a green marker is over them**. Quickly consume it to free space, get bigger and get a large amount of points!

B.3 Controls

B.3.1 Movement

You control the cell by holding the left-mouse button. The cell moves towards the cursor, regardless of how far it is is from the cell (i.e. the speed does not change depending on mouse position). It is recommended to keep the mouse cursor not too far away from the cell, as some processes, such as absorption, are very sensitive to mouse cursor position.

B.3.2 Absorption

As mentioned before, agar can be absorbed at all times, while rigid cells only at times when a green arrow is over them. To absorb an object, the player's cell must push against it with its front part for a short period of time.

When pushing against larger rigid cells, aim for their centroid (it does not necessarily correspond to their kernel). If you have trouble with initiating absorption while pushing against it, try to move away from it and approach it anew.

B.3.3 Commands

- Hold Left-mouse button: Move around.
- Hold Q: Consumes an absorbed object (if present).
- A: Zoom out.
- D: Zoom in.