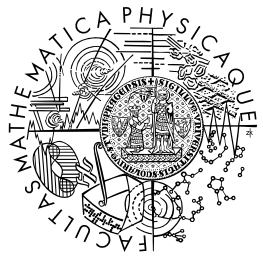

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Marek Dobranský

Efficient simulation of environment destruction in games

Department of Software Engineering

Supervisor of the bachelor thesis: Mgr. Miroslav Kratochvíl
Study programme: Computer Science
Study branch: Programming and Software systems

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 19.7.2017 signature of the author

Title: Efficient simulation of environment destruction in games

Author: Marek Dobranský

Department: Department of Software Engineering

Supervisor: Mgr. Miroslav Kratochvíl, The Department of Software Engineering

Abstract: Destructible environments have become a popular feature of computer games. Currently used game engines employ different approaches to implement such environment. This thesis studies several such approaches and implements some key ideas from available research in a new, combined approach. We use tessellations and boolean operations on triangular meshes to modify rigid-body objects that represent game environment, and create a simple application to demonstrate the approach in a real-time environment. We conclude that the proposed method is mainly suitable for computer games that feature low-polygon meshes.

Keywords: destructible environment, simulation, games, convex decomposition, polygon mesh

I would like to express my gratitude to Mgr. Miroslav Kratochvíl, the supervisor of this thesis, for his patient guidance, feedback and all advice he has given me.

I also want to thank my family and girlfriend for their continued support and encouragement during my Bachelor studies and especially during the time spent working on this thesis.

Contents

Introduction

Destructible environment has been an amusing part of computer games since their beginning, from shooting through the simple bricks in Space invaders to blowing up buildings in the Battlefield series. Current development is trying to achieve highly realistic destruction effects by inventing and improving various simulation approaches of 2D and 3D destructible game worlds.

Despite the advances in the field, the excessive complexity of accurate physical simulation introduces a trade-off: To achieve sufficient performance required to satisfy real-time constraints of the game environment (most notably acceptable and consistent frame rate), the game developers are forced to relax their demands on realism, usually by simplifying the game environment and neglecting less-important aspects of physical simulation.

The goal of this thesis is to review current techniques used to simulate such environments, implement a simple game environment to test some of the approaches, and design and measure an approach applicable in the game environment that is based on a combination of the reviewed techniques.

Related work A large part of the research on the destructible environment is done commercially to be deployed in game titles; the thesis reviews a selection of those approaches. The commercial research is usually based on rigorous academic research of the basic algorithms required for the task, including various tessellations, convex decompositions, methods of physical simulation and geometric representation of solid objects. We specially refer to the work on dynamically destructible rigid-bodies (treated in detail in section 1.2), boolean operations on meshes using Nef polyhedra Bieri [6], an approach based on discrete particle method focused on inner body stress simulation [10], and one based on Voronoi tessellation [15].

Approach As a main result of the thesis, we present an algorithm that computes a fracture of a 3D solid object using boolean operations on 3D objects and Voronoi tessellation. The algorithm is set into the environment of a simple physical simulation of objects of arbitrary shape and used to modify the objects in the case when a critical force is applied to them, i. e. when they are shot by the player or when the simulation detects collisions with other objects.

The algorithm is similar to one described by Müller et al. [15], with some improvements:

- We try to reuse currently available open-source libraries to implement

primitive operations on 3D solids, which has influenced the internal representation of the objects (e. g. the mesh representation used for rendering and model storage is different than the one required to run tessellations) and exact steps of the algorithms (e. g. we use mesh subtraction instead of object reconstruction).

- To hide the latency caused by fracturing-related computation, we offload it to other program threads and insert the result to the main simulation only as soon as the computation finishes. Without the drop in frame rate, the introduced delay is almost negligible from the user perspective.

Finally, the performance of the algorithm is measured on objects with varying complexity.

Thesis structure The thesis is organized as follows: Chapter 1 describes the techniques used in computer games and simulations, and focuses on several results of recent research. Chapter 2 introduces open-source libraries and implementations of the 3D geometric algorithms that are required for the simulation. Chapter 3 illustrates the design of our implementation and presents measurements of its performance. After conclusion we continue with two appendices: Appendix A is a user guide for running the testing program, and appendix B provides some insight into the inner structure of the program.

1. Overview of techniques

This chapter introduces the techniques used to simulate destructible environments. First, we talk about the development of the destructible environment in several games and game engines, then about general approaches to this problem.

We repeatedly categorize the content of the game world in following way: We use the word *object* to refer to any building, crate, door, tree or other items that occur in the game environment, but excluding terrain, distant environment like skyboxes and any in-game characters. Most of the modern 3D games use the term destructible environment as a reference to destructible objects because they do not support the destruction of terrain. We decided to comply with this terminology and, unless specified otherwise, use the term destructible environment as a reference to a destruction of game objects and not the terrain.

1.1 Implementations in mainstream gaming

In this brief overview, we introduce the most common approaches to environment destruction that can be seen through the games released in last 40 years. More common approaches can be seen used in newly released games.

Object replacement or removal was the first [4] method used to simulate a destructible environment in a computer game, mostly because it is straightforward and undemanding. This approach is based on swapping models or any other kind of visual representations, for more damaged ones or completely removing them. Because it relies on pre-made models, exact collision points on the models are not considered, and the result is always the same. If we were to consider N points of taking damage, the number of different models could grow up to 2^N. A large number of models is not practical for game development, other approaches are therefore used for breaking the object at the precise point. Despite its simplicity and limitations, this approach can still produce a very desirable result. In fact, it is still most widely used approach to a destructible environment in computer games.

In our simplified view on 2D games, we do not differentiate between objects and terrain and refer to both as the environment. First 2D games featuring the destructible environment were arcade games like the *Space Invaders (1978)*[1], where the environment is represented by cells in a grid. After taking damage, the visual representation of the cell is replaced by one looking more

[1] `https://en.wikipedia.org/wiki/Space_Invaders`

Figure 1.1: *Source* engine swaps door models. Image taken from *Counter Strike: Global Offensive*. Successively damaging any doors in the game always produces this sequence of models, regardless of the exact point where the damage was applied.

damaged and then finally completely removed. About a decade later, new environment destruction technique was used in games like *Scorched Earth (1991)*[2] and *Worms (1995)*[3]. Collision and removal of terrain in those games is based on individual pixels rather than whole objects, which creates a more realistic visual effect. In *Scorched earth* players typically need to blow away a hill dividing them, in *Worms* it is common to dig a tunnel to hide from your enemies gunfire.

In 3D games, the principal technique of implementation of destructible objects has not changed much over the years. For every object that the player can modify there is a prepared set of alternative models with various amount of damage applied. Based on how much damage is applied, the models are swapped and eventually completely removed, as seen in fig. 1.1. Swapping the models is usually accompanied by animations, debris and dust generation, designed to hide the unrealistic and instant change of the object from the player. The disadvantage of this method is the necessity to replace the in-game objects as a whole — there has to be a significant number of objects pre-made for different scenarios to make the game look realistic. A small number of pre-made damaged models means this approach can not flexibly react to specific player actions. As an example, in the *Source* engine, the hole in the door (fig. 1.1) is always created at the same place, regardless of the point of the impact. Another example can be found in the game *Duke Nukem 3D*[4], where specific parts of the walls are constructed as separate objects that disappear when hit.

Height map approach is closely intertwined with terrain generation.

[2] `https://en.wikipedia.org/wiki/Scorched_Earth_(video_game)`
[3] `https://www.team17.com/games/worms-original`
[4] `https://en.wikipedia.org/wiki/Duke_Nukem_3D`

The basic principle behind modifying the environment with this method is changing the height of the terrain at given point. We can find this method used in the first 3D games that featured destructible terrain, e. g. *Magic Carpet (1994)* [5] or *Starfighter 3000 (1994)* [6].

The height map of a 3D terrain is defined as a uniform 2D grid of values representing the height of their respective column in 3D space. A convenient approach is to use 2D gray-scale bitmap and represent the height as a color distance from the white color, black being the maximum. Because the grid only contains information about the discrete points, interpolation is used to create continuous terrain. We can also view this method as creating a function of two coordinates on the plane giving us the value on the vertical axis. The definition of function forbids more than one function value at each input point. Therefore this method can not represent multiple layers of terrain. The relative simplicity of the approach is counterweighted by the fact that it can only change the height of the terrain and does not allow creating caves, tunnels or similar hollow features.

Geo-Mod (*Geometry Modification Technology*[3]) is an engine developed for the *Red Faction (2001)*[7] video game. It approaches the modification of the terrain by creating objects representing an empty space. After every collision, a new object is created at the point of collision. This new object is then subtracted from the terrain, creating the modified terrain with the newly created hole. The difference of the meshes is calculated in real time. Game reviews suggest that the engine does not work well with the buildings and other objects. *Geomod* represented the first significant attempt to create the fully destructible 3D environment that would work under real-time constraints.

Geo-Mod 2 [3][8] does not feature destructible terrain, instead, it focuses on realistic destruction of buildings. A set of smaller objects is used to represent each building as a ragdoll (multiple objects connected with joints) for the simulation of inner stresses. Switching from using a conventional 3D model to the ragdoll simulations requires at least basic knowledge of civil engineering to properly analyze the structural stability of the building and create a ragdoll model simulating it accurately. This process is therefore done by hand in the development process and can not be modified at a runtime. The game avoids any interaction of multiple buildings requiring simulating their behavior at once because of the high complexity of simulation. This limits the engine in the scale of the game world and mutual proximity of

[5] `https://ultimatehistoryvideogames.jimdo.com/magic-carpet`

[6] `https://en.wikipedia.org/wiki/Star_Fighter_(video_game)`

[7] `http://redfaction.wikia.com/wiki/Red_Faction`

[8] Geo-Mod 2.0 presentation video `https://www.youtube.com/watch?v=lICurOVsNv0`

Figure 1.2: The arrangement of destructible elements for the large-scale destruction in *Frostbite* engine (on the left). Rest of the images shows the process of large-scale destruction in *Battlefield: Bad Company* implemented using *Frostbite 1*.

destructible objects.

Frostbite [9] engine (and mainly its component *Destruction* [2]) is currently used in new mainstream games that feature destructible environment e. g. *Battlefield* series or *Star Wars Battlefront (2015)*. It supports two different kinds of destruction micro-destruction on the surface and the large scale predetermined destruction on whole buildings.

The dynamic micro-destruction focuses on creating small dents into the surface of the object. The dents are created dynamically at the point of impact and can be placed on any point of the surface.

The large-scale destruction is focused on destroying entire buildings. The buildings are created from smaller parts that are linked together. Each part can disappear on its own (fig. 1.2), and when there are not enough parts left, the whole building collapses. It does not use any physical simulation in this process, and therefore structural stability is determined solely by the number of removed parts of the building.

1.2 Methods and algorithms

Here we give a short overview of several more rigorous algorithms commonly used for simulations of destructible environment.

In simulations, we consider two types of objects: soft bodies and rigid bodies. A rigid body represents an object with a constant shape where every two points on the body are always in the same relative position. On the other hand, soft bodies are deformable under an applied force, and any point on the body can change its position independently on every other point. Although there are no actual rigid objects in the real world, soft body simulation is

[9] `http://www.frostbite.com/about/frostbite-3`

computationally more expensive than the rigid body one. Therefore, in the computer games, rigid body simulation is used almost exclusively. Although we do not perform soft body simulations, we try to emulate them in real-time on rigid bodies.

When trying to save a processing power, there is another way to do it on soft bodies. Many soft bodies are non-essential to the gameplay, and their simulation would not bring more value to the game e. g. waving flag or flowing water. Those bodies can be implemented as a pre-rendered animations and not actual simulations.

1.2.1 Soft body deformation

In this section, we shortly introduce two different approaches for simulation of soft bodies. Despite the fact that soft bodies are currently rarely used in computer games to the extent of destructible environment, several soft body objects can show up in a game. We also expect that with an evolution of more powerful hardware, soft body deformation will make its way into the gaming world e. g. *Next Car Game: Wreckfest (2018)* [10] will use soft bodies to simulate car crashes.

Finite Element Method (FEM) is a numerical method used to simulate the behavior of a system that can be modelled by solving the behavior of the smaller discrete parts, called finite elements. Each element calculates its physical state, e. g. stress or temperature, and propagates the results to neighboring elements. This model can be used for simulation of fluid dynamics, brittle fractures [16], ductility [18], elasticity, heat transfer and other physical properties. It is beneficial in engineering, modelling and rendering scenes for computer generated images [5].

FEM requires a lot of computational resources and is mostly used in simulations that are not under real-time constraints. The algorithms like O'Brien and Parker [17] propose, that optimised version of FEM can be used in computer games.

Material point method (MPM) is a numerical method for modelling an object as a continuum mass. Continuum model assumes that the object fills the space it occupies and is not a set of discrete particles. MPM simulates such materials by using two different views of the data, material points (Lagrangian elements) and Eulerian grid. MPM is a particle method but thanks to the Eulerian grid it can be applied to continuum materials.

MPM is more computationally expensive than FEM, as the grid must be reset at the end of each MPM calculation step and reinitialized at the begin-

[10] `http://store.steampowered.com/app/228380/Next_Car_Game_Wreckfest/`

ning of the following step. On the other hand, MPM is a meshfree method and does not require remeshing steps and is less susceptible to numerical errors.

MPM works by projecting the data from the particles to the mesh, determining and applying the velocities on the nodes of the mesh and then updating the particles based on the deformation of the mesh. After each loop, mesh needs to be reset while the data are stored in particles.

A simplified overview of algorithm steps follows, for more details see the thesis of Jiang [11].

1. Grid data are reinitialized to default values.
2. Weights and weight gradients are computed on every particle.
3. Mass and momentum are transferred from the particles to the grid.
4. The explicit forces on nodes are calculated
5. The explicit nodal velocity update is performed
6. Grid based collision is performed on vertices.
7. Particles are updated from grid velocities.

These steps are illustrated in fig. 1.3.

MPM is useful for both fluid and soft body dynamics. It can simulate deformation, fractures, heat transfer, melting and other changes of the state of an object.

A popular example of MPM deployment can be seen in snow simulation in the *Disney* film *Frozen (2013)*. The simulation software *Matterhorn*[11] computes the behavior of different types of snow (e. g. wet, fresh, sticky) and other materials, such as sand or mud.

1.2.2 Rigid body decomposition

In the soft body, application of the force propagates across the particles, and the connected particles break apart when the limit is exceeded. This creates the fracturing that may lead to the whole body splitting. In the rigid body simulation, the body has no internal structure, and therefore the applied force can only change the momentum of the entire body. This prohibits the deformation or fracturing a rigid body in a simulation.

[11] `https://www.disneyanimation.com/technology/innovations/matterhorn`

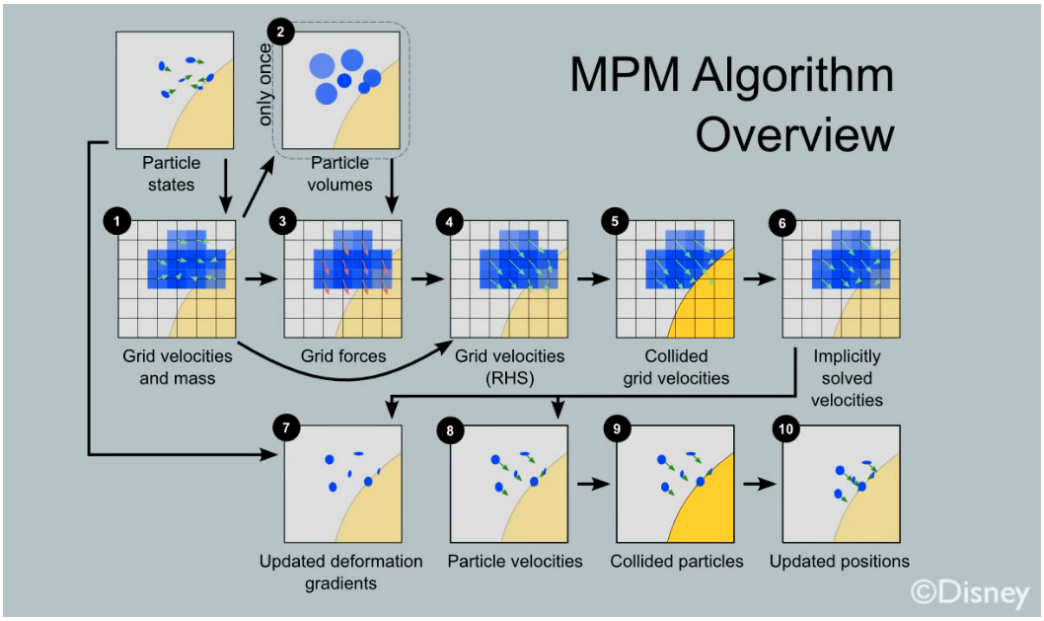

Figure 1.3: Material point method algorithm overview. The top and the bottom rows operate in particle domain (Lagrangian) while the middle depicts grid-based (Eulerian) operations [20].

To solve this problem, there are numerous approaches of decomposing the rigid body in the way that imitates fractures achieved by exceeding the elasticity of the soft material. The most common approach is a decomposition of a rigid body into smaller parts. Some of the common methods for decomposition are: slicing by planes [14], convex decomposition [12], tetrahedralization [9] and Voronoi tessellation. In this section, we describe a process of Voronoi tessellation.

Voronoi tessellation is a method of decomposing a solid object into smaller parts, as shown on fig. 1.4. It is also applicable for e. g. terrain generation [19], but we focus only on object decomposition. Assuming the input is a closed triangular mesh with non-empty volume, the tessellation can be done in following three steps:

Delaunay tetrahedral decomposition Given points P in general position (the vertices of input mesh and a set of points inside its volume), tetrahedral mesh $DT(P)$ can be generated satisfying the following condition: no point in P is inside the circumscribed sphere of any tetrahedra in $DT(P)$ [8].

Creating Voronoi diagram For a Delaunay tetrahedral decomposition,

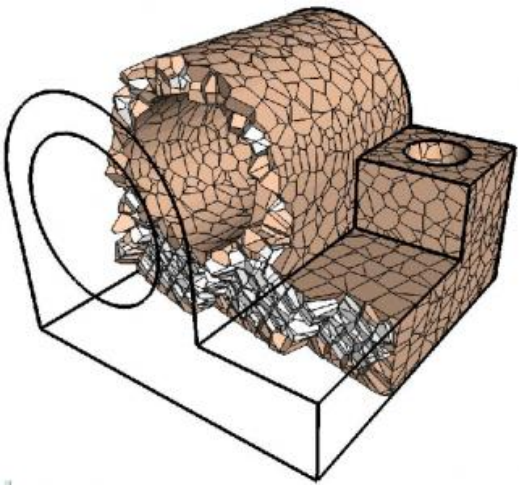

Figure 1.4: The result of Voronoi tessellation by Yan et al. [21]

its dual graph (with vertices in the center of tetrahedrons circumscribed sphere) is a Voronoi diagram (see fig. 1.4).

Clipping the Voronoi diagram Boundary cells of the Voronoi diagram are infinite (see fig. 1.5) and need to be clipped by the original input triangular mesh. The efficient algorithm proposed by Yan et al. [21] for this task finds the intersection of the boundary Voronoi cell with the triangular mesh and continues with neighborhood propagation to determine all intersections.

As we can see in the performance diagram fig. 1.6 the Voronoi tessellation with hundreds of thousands of Voronoi sites can take a number of seconds to calculate. However, for a destructible environment, hundreds of pieces will suffice. Therefore this method can be used in a real-time application.

1.3 Related research

In this section, we review two results of recent research that propose different techniques for the environment destruction in computer games. The first technique focuses on simulating the forces inside the object with the approach based on particle methods. The second one focuses solely on the destruction of rigid bodies with the use of Voronoi decomposition.

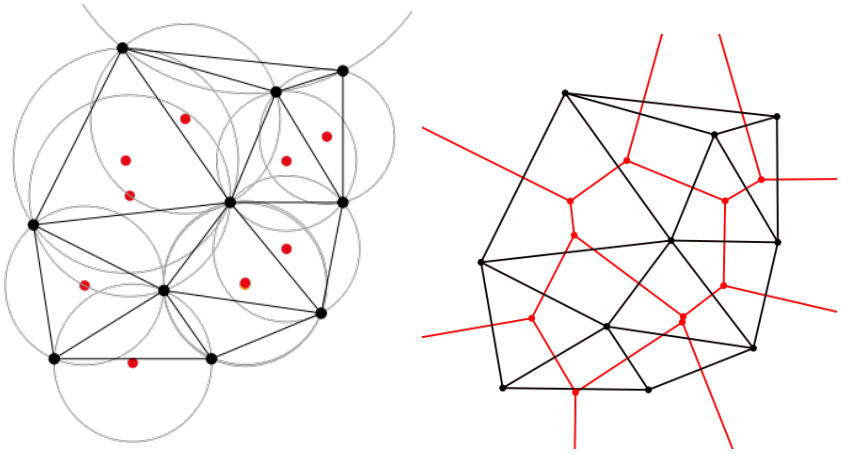

Figure 1.5: Transformation of 2D Delaunay triangulation to Voronoi diagram. Source: `https://en.wikipedia.org/wiki/Delaunay_triangulation`

1.3.1 A fast method for simulating destruction and the generated dust and debris

The method of Imagire et al. [10] approaches destruction on three scales. At first, the destruction is performed on a coarse scale, and then the applied energy is used to calculate the amount and size of smaller debris and finally dust particles.

Destruction on a coarse scale , on this level the authors use Extended Distinct Element Method (EDEM) which is based on Element Method described in section 1.2.1. Instead of using particles, EDEM uses a set of rigid bodies that need to be produced by some tessellation method. The extension over traditional DEM is the use of pore springs to connect the adjacent elements (see fig. 1.7). The authors compare the elements to individual bricks (the distinct elements) held together by layers of mortar (the pore springs). If the applied force is sufficient to move the elements far enough from each other, the fracturing process is initiated. Pore springs also help the object retain its original shape after the force is applied.
Algorithm for creating EDEM elements

1. Represent the original object as a closed surface model.
2. Arbitrarily arrange the EDEM elements inside the object. Elements are allowed to overlap at this point.

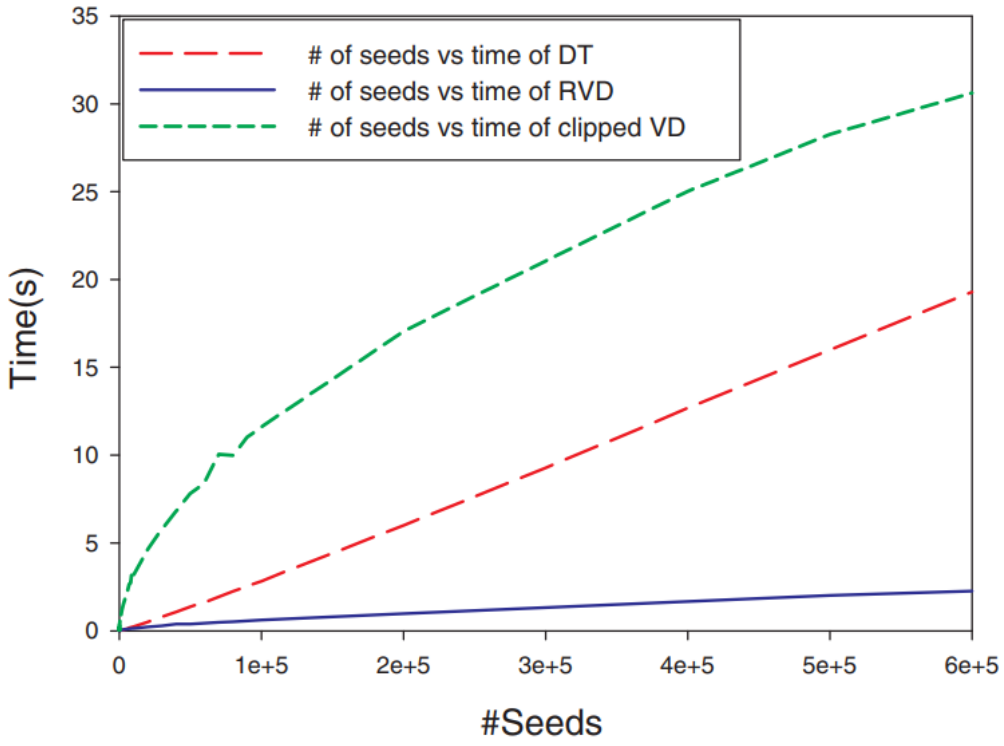

Figure 1.6: Timing curve of clipped Voronoi diagram computation against the number of Voronoi seed points, on a model with 1000 boundary triangles. To achieve clipped Voronoi diagram all three tasks must be completed. Source: Yan et al. [21]

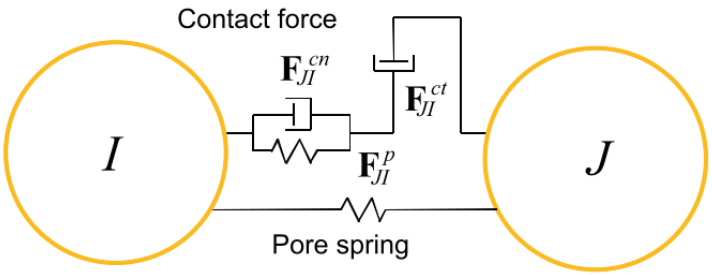

Figure 1.7: The force between two EDEM elements. Source: Imagire et al. [10]

3. Move elements by performing the EDEM simulation. Only the contact force is considered in this simulation.
4. Perform collision detection between the object's surface and elements, making sure that they always stay inside the object.
5. Repeat (3)–(4) until elements are stabilized.
6. Construct a Delaunay diagram from the set of elements and put the pore springs on the Delaunay edges that connect the elements

We can see the distribution of the EDEM elements inside objects after the collision on fig. 1.8.

The position $\mathbf{x}_I$ and velocity $\mathbf{v}_I$ of the element I can be found using Newton's equation of motion as follows:

$$M\frac{d\mathbf{v}_I}{dt} = \sum_{J\in contact} \mathbf{F}_{JI}^{c} + \sum_{K\in pore} \mathbf{F}_{KI}^{p} + M\mathbf{g}$$

$$\frac{d\mathbf{x}_I}{dt} = \mathbf{v}_I$$

Here, $\mathbf{g}$ is the gravitational vector, $\mathbf{F}_{JI}^{c}$ is the contact force, $\mathbf{F}_{KI}^{p}$ is the force due to the pore springs and M is the element's mass. Two elements {I, J} are in contact, if they are closer to each other than the diameter of a single element.

Fine debris generation and simulation If a fracture between EDEM elements happened, we can determine if there was enough energy to break EDEM element into smaller debris. The probability distribution is used to determine the size of debris. Debris is taken out of EDEM simulation and put into particle simulation, where each piece is represented as a particle without volume.

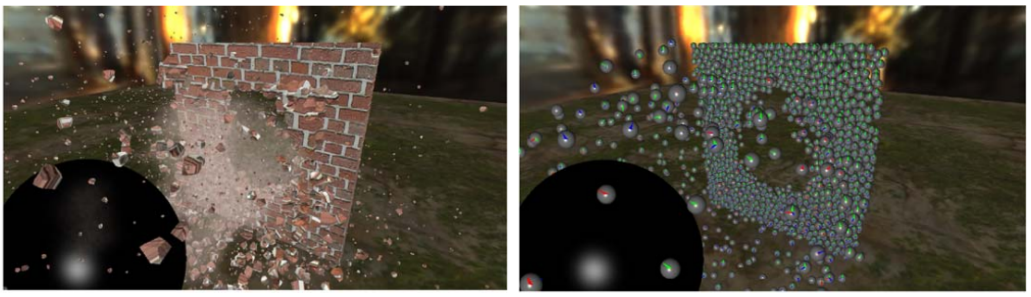

Figure 1.8: View of final rendering with generated dust and debris (left) and equivalent view showing underlying EDEM elements (right). Source: Imagire et al. [10]

Dust generation and simulation Amount of generated dust is based on fracture energy and results of debris generation. Instead of simulating particles smaller than predetermined margin, they are represented as dust in a grid-based fluid simulation. The density of particular cell represents the amount of dust.

EDEM elements	FPS
128	320
256	160
512	75
1024	30
2048	9.1

Table 1.1: Performance of EDEM element method without rendering [10]

In conclusion, based on table 1.1, the frames per second (fps) drop proportionally with growing number of EDEM elements. The method achieved interactive fps on moderate number of EDEM elements. In addition, the proposed debris and dust generation can be combined with other methods.

1.3.2 Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions

The approach of Müller et al. [15] does not try to simulate the internal forces of an object and focuses only on rigid body decomposition. The idea behind this method is to represent the mesh as a compound shape of convex parts.

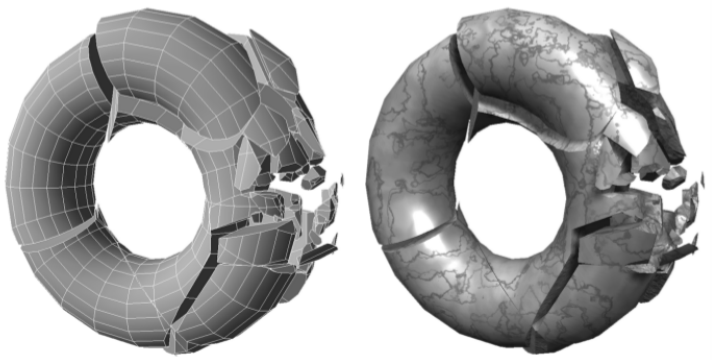

Figure 1.9: The decomposition to the initial compound mesh does not become visible when a fracture pattern is applied. Source: Müller et al. [15]

The algorithm used for convex decomposition is Volumetric Approximate Convex Decomposition (VACD), which works by introducing the Voronoi decomposition into a bounding box of the mesh and then clipping the Voronoi cells by the mesh. To determine the decomposed shapes, we use a prepared fracture pattern. The fracture pattern is pre-computed and represented as a set of convex cells. The algorithm works as follows (the process is visualized in fig. 1.10:

1. The fracture pattern is aligned with the point of impact, and rotated and scaled randomly to avoid an occurrence of same-looking patterns.

2. The intersections of all cells with all convex parts are computed. To compute the intersection of a single cell with a single convex part, the convex part is clipped against all the planes of the cell one by one. At the end of this step, we have a set of new convex parts, and each convex part belongs to exactly one cell.

3. If there are convex parts that together entirely fill one cell, we can combine them into single new one convex part. The test is carried out using a simple volume comparison.

4. All convex parts that belong to one cell are combined to form a new compound part. This ensures that the temporary parts of decomposition into compound shape are not visible after the fracture (see fig. 1.9).

5. Finally, the separate islands of convex parts are detected and individual compound shapes are constructed for them.

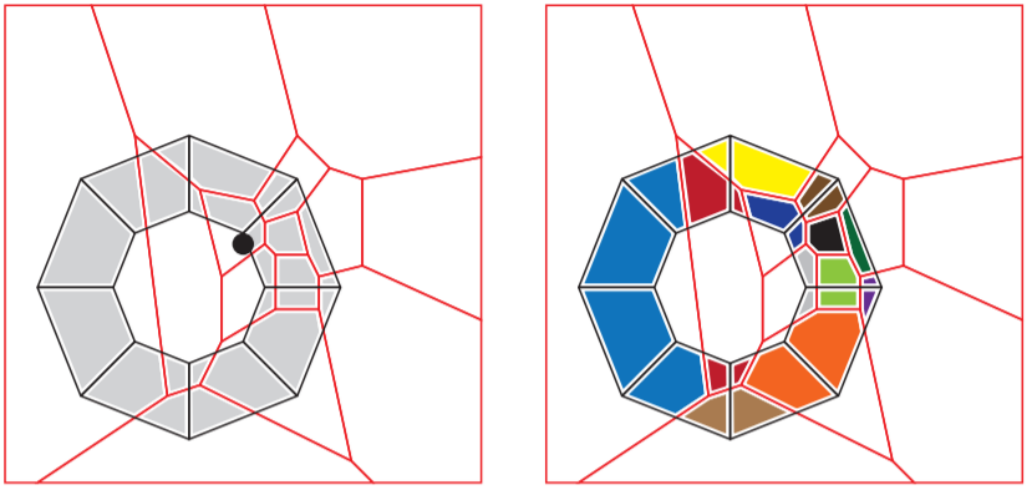

Figure 1.10: Overview of the fracture algorithm. Left: The fracture pattern (red) is aligned with the impact location (black dot). Middle: All convex pieces are intersected with all cells. The green convex pieces can be welded to form a single piece because they cover the entire cell. Pieces within one cell become a new compound part (same colors). Island detection finds that the dark red compound needs to be split. Source: Müller et al. [15]

The cost of fracturing in this method depends only on the number of mesh triangles of the object being fractured and the resolution of fracture pattern. The paper suggests that the model with 10^6 vertices and $5 \cdot 10^5$ faces can be fractured under 50ms. This makes this method suitable for real-time computer games.

2. Review of related software libraries

In this chapter, we review some software libraries that can be used for implementing the algorithms required for constructing a destructible environment. We do not talk about all-in-one software development kits with their approach to a destructible environment already implemented. Instead, our focus is on building a game with our own approach to the destructible environment.

To build a game, we will surely need a physics engine, a graphics engine, and a selection of geometric libraries to support the computation of the destructible environment. We mention which libraries we used in the implementation and details of their use can be found in appendix B.

2.1 Physics engine

Physics engines are used in games to provide a simulation of applied forces to in-game objects and collision detection. Many physics engines are directly bundled with rendering, audio engines to create a complete game engine.

To support our implementation of destructible environment we need a physics engine to detect collisions, move the user-controlled vehicle and simulate gravity and mechanical forces. There are no special needs for the physics engine in our applications that would not be supported in most engines. Some of the commonly used engines are Open Dynamics Engine, Newton Game Dynamics, Bullet, Simulation Open Framework Architecture, Tokamak. All of them provide the simulation of rigid bodies and collision detection. Other features are not relevant to us. Therefore we decided to used Bullet physics on the base of author's previous experience.

2.2 Geometric libraries

Here we preview some of the libraries focused on a decomposition or modification of 3D meshed objects. The preview is focused on three techniques used in destructible environment, boolean operations, Voronoi tessellation and convex decomposition,

2.2.1 Boolean operations

When searching for efficient library able to compute the difference of two 3D triangular meshes, we found out that the majority of software utilizing geometric library is dependent on using **The Computational Geometry Algorithms Library** [1] or shortly **CGAL**. CGAL provides polyhedral surfaces that are closed for Boolean operations, and it is possible to convert data between meshes and polyhedrons.

The mentioned polyhedral surfaces or Nef polyhedrons [6], require the input mesh to be an orientable 2-manifold (for more details on this data structure see appendix B). Nef polyhedron works with two data structures, one that represents the local neighborhoods of vertices, which is in itself already a complete description, and a data structure that connects these neighborhoods up to a global data structure with edges, facets, and volumes. This redundancy in data makes Nef polyhedron a large data structure that is not optimized for fast construction and not the most suitable for real-time deployment.

We also considered using a **Cork Boolean Library** [2] but we did not find it to be as robust as CGAL and encountered problems and instability on valid data.

2.2.2 Voronoi tessellation

We learned that to Voronoi tessellation can be beneficial to either creating a compound body held together by springs or used as a tool for subtracting parts of the mesh. Here are some libraries useful for both tasks.

CGAL also includes package providing different 3D triangulations, mainly Delaunay triangulation and the possibility of creating Voronoi diagram as its dual graph. However, CGAL does not provide means to clip the Voronoi cells against the surface mesh.

Voro++ [3] is a library for carrying out three-dimensional computations of the Voronoi tessellation. It calculates Voronoi cell for each particle individually and is suited for high-performance calculations on large scale particle systems. It is also able to clip Voronoi cells to any user defined boundary. This library would be well suited for decomposing entire objects into Voronoi cells.

[1] `http://www.cgal.org/`

[2] `https://github.com/gilbo/cork#cork-boolean-library`

[3] `http://math.lbl.gov/voro++/`

Qhull [4] provides the means to compute convex hull, Delaunay triangulation, Voronoi diagram in 2,3 or 4 dimensional space.

Because implementation requires only one Voronoi cell per collision, we chose to use the simpler Voro++ library for this task.

2.2.3 Convex decomposition

Convex decomposition is critical to fast and accurate collision detection on 3D meshes. We introduce two libraries with different approaches to the task.

CGAL provides the means for decomposing the polyhedral shape (triangular mesh can be converted into polyhedron) into a set of convex polyhedrons. CGAL is aiming for an exact convex decomposition, which produces a large number of small pieces [5].

Hierarchical Approximate Convex Decomposition or HACD [13] is a library and algorithm providing a convex decomposition that approximates the original shape. The approximation yields fewer pieces and is faster than exact calculations, but the result is not exact. For the reasons that we are implementing a destructible environment in a computer game, we can allow for deviations in the shape of the objects from their respective visual representations. Those deviations should be too small to be noticed in the game. This makes us choose HACD over CGAL.

Convex Decomposition Library [6] produces approximate convex decomposition. It is an older library that is currently marked deprecated in favor of HACD.

We can see the difference in the result of exact and approximate approaches on fig. 2.1. It is evident that we do not want to use the exact decomposition in a computer game, but on the other hand for the simulations where the calculation time is not critical, the approximations should not be used.

[4] `http://www.qhull.org/`

[5] `http://doc.cgal.org/latest/Convex_decomposition_3`

[6] `http://codesuppository.blogspot.cz/2009/11/convex-decomposition-library-now.html`

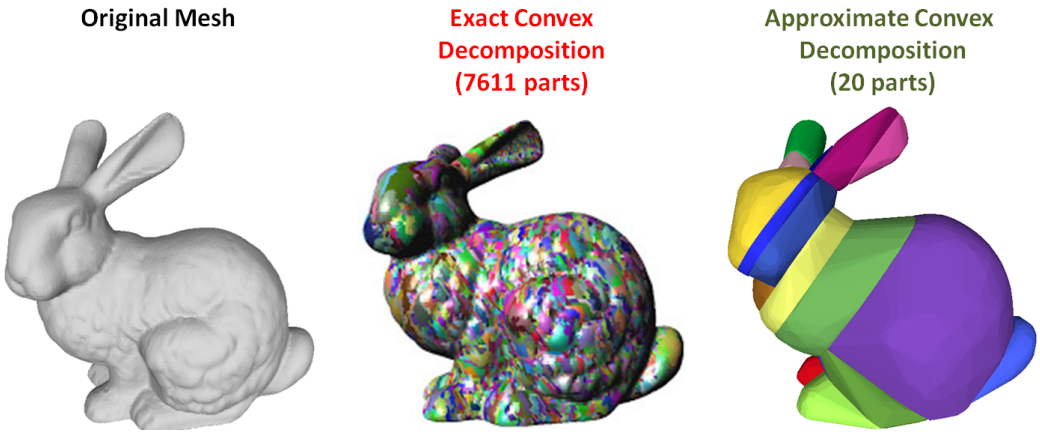

Figure 2.1: Difference between original mesh, exact convex decomposition and an approximate convex decomposition. Source: `https://github.com/kmammou/v-hacd`

3. Implementation

In this chapter, we describe the implementation of our demonstration game and explain the basic principles and algorithms behind it. Then we present measurements of performance of main algorithms used in the implementation.

At first, we considered implementation based on *A fast method for simulating destruction and the generated dust and debris* (FMSDGDD) (see section 1.3.1). However, after degrading performance issues section 3.5 we decided to abandon this approach.

After consideration of various approaches implemented in games and also proposed efficient solutions to the problem of real-time destructible environment, we decided to implement and test an approach based on *Geomod* (described in section 1.1) technique. The *Geomod* inspired us to create an object representing an empty space and use the boolean subtraction operation on the mesh to generate a damaged body. The abandoned FMSDGDD approach influenced us to generate debris equivalent to the removed volume. We implemented this idea by using an intersection of *Geomod* inspired empty space and original mesh as debris.

3.1 Main algorithm

As mentioned, our algorithm uses boolean operations similarly to *Geomod*s removal of an empty space from the terrain. The difference is that we apply this method to rigid body objects and permanently alter their mesh. The shape of removed object is determined by Voronoi cell that is generated dynamically for every collision. The collision information is received from the physics engine.

Our approach generates Voronoi cell at the point of collision. The decision to use one cell instead of a fracture pattern, like the one described in Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions algorithm (see section 1.3.2), is based on the fact that we want to observe how dependent is our design on the complexity of destructed objects and not on the complexity of fracture pattern. Our implementation is easily expandable to the use of multiple cells or cells of any shape.

After the cell generation, the difference of original mesh and the Voronoi cell is calculated to represent the damaged objects. To generate the debris, the intersection of original mesh and the Voronoi cell is calculated. This action effectively cuts the object into two or more pieces, all of which are put back into simulation and can be damaged again (see fig. 3.1). The

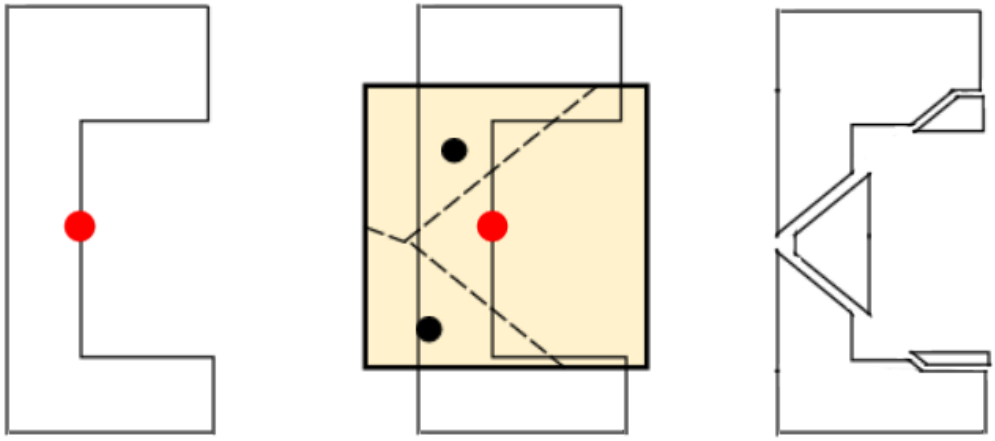

Figure 3.1: object with point of collision (left), generated Voronoi cells (centre), object divided into five new smaller objects after subtraction of the Voronoi cell belonging to the point of collision (right)

Voronoi cell was chosen because it has easily randomizable shape and provides aesthetically good results.

The cost of fracturing in our implementation is solely dependent on the size and complexity of the fractured object section 3.5. This makes the method suitable for use in computer games with a large number of simple objects with low polygon meshes.

To generate the Voronoi cell, we create a closed domain with the center at the point of collision. In the domain, we need to create random points, so the Voronoi cell of the point of collision does not cover entire domain. This step also ensures variability of generated cells. After that, we can use a Voronoi cell belonging to the point of collision as the mesh we subtract from the object being damaged. The boundaries of the domain clip the generated Voronoi cell, therefore the Voronoi cell can never be larger than the domain. Randomization of the size and shape of a Voronoi cell guarantees different result after every damage application.

3.2 Program Structure

The program is structured into four main components, Physics, Graphics, Controllers and Mesh Manipulation (see fig. 3.2). Physics performs rigid body simulation and detects collisions. Graphics is used as an output device. Controllers contain the core of the application and process user input. Mesh

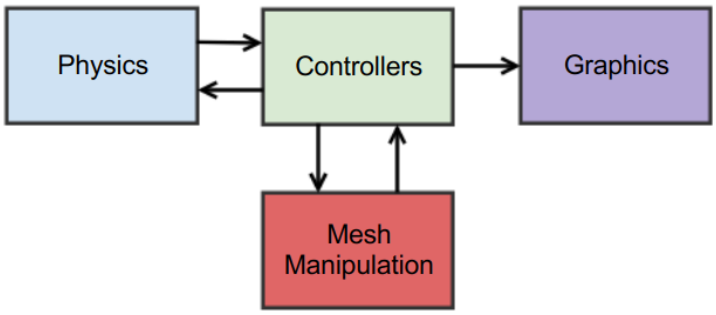

Figure 3.2: Diagram showing architecture of the application

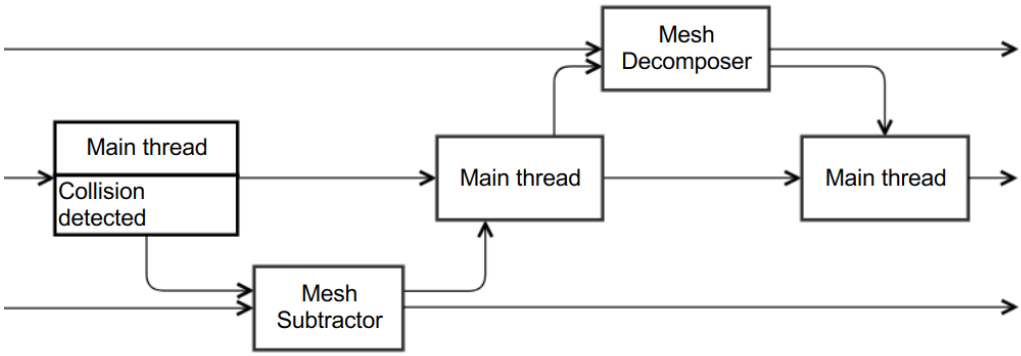

Figure 3.3: Diagram is showing multiple threads handling collision event.

Manipulation is the part of the application that handles the destruction of the environment.

The main program loop runs in following steps:

1. Perform a step in physics simulation.
2. Handle collisions and perform destruction, as described in section 3.3.
3. Read user input and then apply correct forces to the controlled vehicle.
4. Render the current state of objects. In this step, a graphical representation of every object is updated to comply with its rigid body version.

We want to keep a constant frame-rate in our application, but we do not expect the mesh subtraction and convex decomposition tasks to perform in required time, and therefore we execute them asynchronously in separate threads. As a result, the program is running in three threads (fig. 3.3): the main thread, a thread for subtracting meshes and a thread for decomposing

triangular mesh into a set of convex shapes (section 3.4). Both subtraction and decomposition threads communicate solely with the main thread, and all communication is done in producer-consumer model. Figure 3.4 shows the changes of the object and its collision shape across all threads.

We use only one thread for all mesh subtraction tasks because we anticipate that the most of the consecutive collisions are going to be triggered by the same object — shooting at one building multiple times in a row. In this situation, one subtraction does not have valid input data until the previous one has finished, which leads to sequential processing. If we were to expect the collisions are occurring randomly on independent objects we could implement a thread pool for resolving subtraction tasks. When processing multiple collisions on the same object with multiple threads, the order of applying subtraction is not important as long as all tasks are processed and applied.

A use of a larger thread pool could be useful for convex decomposition. With multiple threads, the object could have been changed since the current calculation started. This conflicting state means we can not apply the current result, but we know for sure that another decomposition task was created when the change occurred. We do not know whether the newer task has already finished or not, but we know for certain, that if we discard our decomposition, the object has either temporary shape or a new valid decomposition. We did not implement the thread pool solution because current common gaming hardware is not well suited for running a large number of simultaneous threads.

3.3 Collision handling

After the detection of the collision, physics engine gives us a reference to two rigid bodies participating in that collision, point of collision and vector of applied impulse. For simplification, we consider only one object, the point of impact and the force. The second object is processed symmetrically.

Some collisions can be results of an object placed on ground or collisions with not enough force to damage the object. We need to filter out those unwanted collisions.

For every collision that is selected as acceptable for damaging the object, we generate Voronoi cell as described before, and move meshes of both objects into a task for mesh subtraction thread. After enqueuing all collisions, we check if there are any prepared subtraction results for further use. The result of one subtraction task is a set of meshes that represent new objects. For every mesh, we create a new incomplete object without its convex decom-

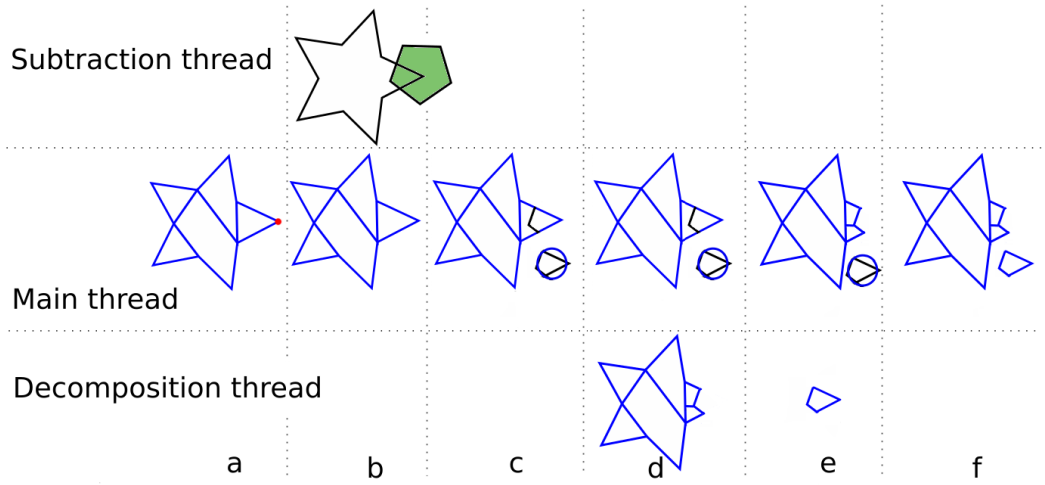

Figure 3.4: Simplified overview of collision handling across multiple threads in simultaneous time slots. (a): collision was detected (red), (b): Voronoi cell (green) is being subtracted from the original mesh (black) in subtraction thread, (c): the result of the subtraction are two objects — one one without change in collision shape (blue) and one with temporary spherical shape, (d) and (e): new collision shapes are computed in decomposition thread, (f): the final result.

position needed by the physics engine to perform accurate collision detection. Because the decomposition can take relatively long, we create a simple temporary collision shape. We decided to use a sphere for the new objects and keep using the original collision shape that is already available for the original object. Then we create tasks for decomposition of current meshes into new collision shapes and proceed with simulation, not waiting for the result. Decomposition is done in a separate thread and the result is returned to the main thread where decomposition results are applied to objects — temporary shape is replaced by compound shape consisting of convex parts.

This process guarantees that we do not wait for either subtraction or decomposition and therefore we can have stable fps in our game — gameplay experience is highly dependent on stable fps and simulation delayed by a few frames is usually not recognizable and can be covered in animations of dust. To ensure consistent behavior of new objects (we can put them into the simulation, and they do not fall through each other or otherwise not comply with laws of physics) when using a temporary collision shape, we need to make the temporary shape resemble the mesh as much as possible. For the already existing objects keeping the older collision shape for a while longer should not disturb the simulation as the closest objects to this space

are a newly generated object, those should be thrown away from the point of collision either way. For the new object, we use the sphere with the diameter equal to the shortest edge of the objects bounding box, meaning that the newly created objects have smaller collision shapes than meshes. The implementation showed that this factor does not visually impact simulation and the presence of the collision shape ensures that the object does not fall through other objects.

3.4 Convex Decomposition

Regardless of used physics engine, our objects are represented as triangular meshes. Implementing mesh to mesh collisions is possible, but highly impractical. Even if checking every vertex of one mesh against all vertices of second mesh is sufficient, the complexity of algorithm would be dependent on the number of vertices.

To be able to perform mesh to mesh collisions with complexity independent of triangle count, we must find a way to describe the object as a set of geometrically simpler shapes. The convex shapes are the easiest for detecting mutual intersections, but encapsulating whole mesh into a convex hull would produce imprecise collisions. This problem is solved by performing a convex decomposition. Convex decomposition process splits the input object into a set of convex shapes, forming a compound shape. Now the complexity of the collision algorithm depends on the number of convex parts. The size of convex decomposition is dependent on the number of concave features on decomposed mesh [1].

While the exact convex decomposition can still produce a significant number of convex parts [7], in the setting of a computer game, the speed of calculation is much more relevant than the precision — small differences between collision shapes and visual meshes are not considered to be a problem. To be able to perform collision detection at real-time, many approximate convex decomposition algorithms that sacrifice some precision to gain performance have been proposed. One of those algorithms is *Hierarchical Approximate Convex Decomposition* algorithm (see section 2.2.3) which we decided to use.

3.5 Measurements and experiments

In this section, we show the conducted experiments with the goal of evaluating efficiency our approach.

To gather the data, we added special code that dumps the timing from

Model	Count	Triangles
`media/building.obj`	7	60
`media/missile.obj`	1	142
`media/ship.obj`	1	104

Table 3.1: Objects and their complexities used in the measurement.

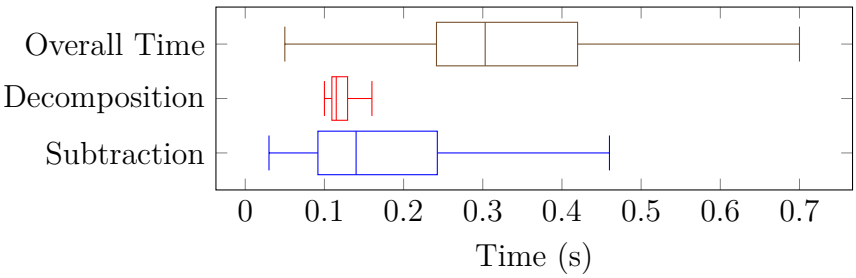

Figure 3.5: Box plot showing the distribution of the duration in seconds of given task (vertical axis)

the program, and we also prepared input files for testing. We are interested in the times of two most complicated tasks in our application, mesh subtraction and convex decomposition.

Data are dumped into *data/* folder. Three files are present in the folder. One contains the dump of the times it took to resolve subtraction tasks, one does the same for decomposition tasks, and the third one contains total times it took for the single object to get from being put into subtraction queue until new decomposition is applied.

We designed two different experiments. The performance test to measure the behavior of our approach in the world with multiple objects and fast sequences of collisions. And the mesh complexity test to identify the relationship between a number of triangles and subtraction/decomposition.

3.5.1 Performance test

For a performance test we prepared a world configuration (copy in *performance_test.cfg*) consisting of objects with models, counts and geometric complexity specified in table 3.1. We recorded about 2000 collisions by repeatedly shooting the buildings at random locations. Recorded data can be seen in fig. 3.5.

Our expectation is that a delay shorter than 300ms between the collision and rendering of its resulting objects would be acceptable for a game and that the separate threads would ensure that this delay would not impact the

# triangles	subtraction	decomposition
12	0.037	0.118
108	0.069	0.135
588	0.134	0.261
2700	0.386	2.362
10092	1.211	12.007

Table 3.2: Average time (in seconds) of processing the given task with the given number of triangles in the mesh. All data are collected on the same cube with only difference in number of triangles.

frame rate. The experiment confirms that on given input we can successfully meet this expectation.

The problem seems to be a high variance in overall time of the whole process starting with mesh subtraction and ending at applying convex decomposition. This can be explained by the tasks waiting in the queue for processing and shows that our solution is not well suited for a fast sequence of collisions. The problem here is that after every collision only one subtraction task is created, but the number of decompositions is nondeterministic and depends on the shape of the destructed object, the generated Voronoi cell and the position of the collision. Use of a thread pool with more advanced bookkeeping of the decomposition dependencies would aid in solving that problem (proposed in section 3.2).

3.5.2 Performance impact of mesh complexity

We designed an experiment to measure a relationship between a number of triangles and time required to complete the two already measured tasks. The collected data are subtraction time and convex decomposition time. For the experiment, we prepared five cubes with the same size but with a different number of triangles in their mesh. We created an isolated environment with only one cube placed in it and then executed the application to collect the data (example configuration in *cube_test.cfg*). The data were generated as a result of a collision after the cube dropped onto the ground. The experiment was repeated with each of the prepared cubes and run multiple times to collect a large sample for statistical analyses. The results of the experiment are shown in table 3.2 .

The experiment showed the limits of our approach. For the mesh subtraction, we can tolerate meshes with the size of about 2000 triangles (see fig. 3.6). On the other hand, as shown on fig. 3.7, convex decomposition times grow significantly faster and reach the critical 300ms somewhere around 1000

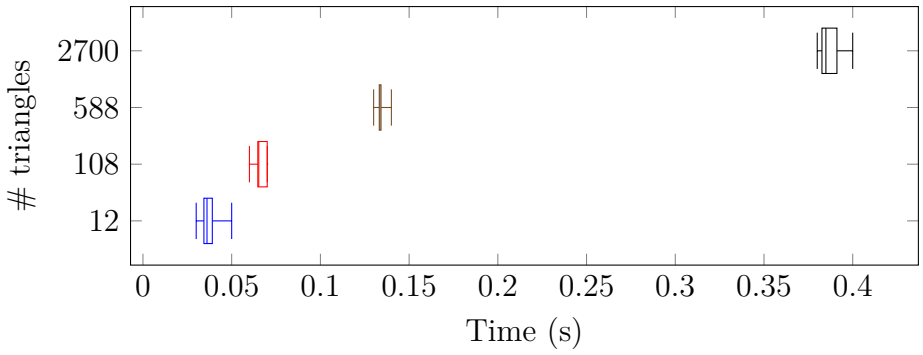

Figure 3.6: Relation between number of triangles and time required for subtraction task.

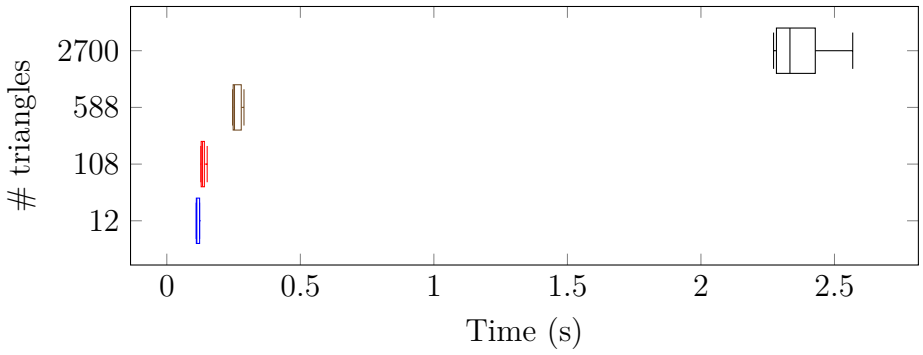

Figure 3.7: Relation between number of triangles and time required for convex decomposition task.

triangles. The measurement for 10092 triangles is omitted from the plots for scaling reasons.

3.5.3 Testing the concept of EDEM

To test EDEM (described in section 1.3.1), we set up a cube divided into 439 tetrahedrons. After introducing constraints to hold the tetrahedrons together, we experienced a drop from 60fps (set as an upper limit) to 13fps. Having a large number of elements connected with springs in the simulation can also trigger an undesirable behavior, such as contractions, retractions and self-induced rapid disassembly of the object. Performance issues and problems with keeping elements in a stable state concluded that the approach was not suitable for our implementation.

Conclusion

This thesis has reviewed currently used techniques used to simulate destructible environments, implemented a simple game environment to test some of the approaches, and designed and measured an approach based on a combination of the reviewed techniques:

- Apart from the technique review in chapter 1, we have added a short overview of the currently available open-source software libraries that may aid the implementation of similar simulations in chapter 2.
- In chapter 3, we have successfully presented an approach that combines boolean operations on 3D objects with Voronoi tessellation and multi-thread computation offloading. The performance experiment in section 3.5.1 show that the visual experience is not impacted by the computation, even though the result of the offloaded computation is presented with a delay.
- The experiments have concluded that the used technique is viable for low-polygon meshes. The convex decomposition, which was the most time-consuming task in the process, creates a noticeable delay with meshes larger than around 1000 triangles. Exact measurements are available in section 3.5.2.
- We have tested the approach of Imagire et al. [10] in the same setting as our implementation. The result in section 3.5.3 strongly suggests that this approach is applicable only to objects divided into a very small number of elements.

Future work

The implementation showed several points that might be viable as starting points for future research:

- We have used CGAL library to perform boolean operations on 3D meshes, the approach it implements has performed consistently. However, CGAL is a geometric library, and it is designed to provide a rich interface with multiple views on data. For the purpose of destructible environment, we propose a library for boolean operations on 3D triangular meshes optimized for use in real-time environments, possibly simplifying and minimizing most other aspects of the library.

- The implementation exhibits a problem with centers of gravity, which are misplaced for some dynamically created concave meshes. Correct computations of centers of gravity for such objects could be added to `Bullet` engine.

- We have not implemented texture mapping for the simulated objects. The generation of correct texture coordinates for the fragments of original objects would benefit the realism of the result.

- It would be interesting to perform independent stability analyses of the in-game objects that would be able to e. g. correctly cause the demolition of an object in which a large, massive part is held in place only by a tiny support. Scanning the environment for unstable objects by exploratively running such tests can be easily performed in separate threads, i. e. possibly not impacting the performance of the main simulation.

- The collisions of controlled vehicle with the environment are currently processed as basic simulation of two colliding rigid bodies. This can produce an out of control spinning and flipping that creates an undesirable visual effects. The application would benefit from adjustment of this part of the simulation.

Bibliography

[1] Convex decomposition of polyhedra.

[2] Destruction. http://battlefield.wikia.com/wiki/Destruction. Accessed: 2017-05-17.

[3] Geo-mod. http://redfaction.wikia.com/wiki/Geo-Mod. Accessed: 2017-05-17.

[4] History 101: Destructible environments in videogames. http://www.gamernode.com/history-101-destructible-environments-in-videogames/. Accessed: 2017-06-17.

[5] Adam W. Bargteil, Chris Wojtan, Jessica K. Hodgins, and Greg Turk. A finite element method for animating large viscoplastic flow. *ACM Trans. Graph.*, 26(3), 2007.

[6] Hanspeter Bieri. Nef polyhedra: A brief introduction. In *Geometric modelling*, pages 43–60. Springer, 1995.

[7] Bernard Chazelle. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. *SIAM Journal on Computing*, 13(3):488–507, 1984.

[8] Paolo Cignoni, Claudio Montani, Raffaele Perego, and Roberto Scopigno. Parallel 3d delaunay triangulation. In *Computer Graphics Forum*, volume 12, pages 129–142. Wiley Online Library, 1993.

[9] Pascal J Frey, Houman Borouchaki, and Paul Louis George. Delaunay tetrahedralization using an advancing-front approach. In *5th International Meshing Roundtable*, pages 31–48, 1996.

[10] Imagire, Takashi, Henry Johan, and Tomoyuki Nishita. A fast method for simulating destruction and the generated dust and debris. *The Visual Computer*, 25.5-7:719–727, 2009.

[11] Chenfanfu Jiang. *The material point method for the physics-based simulation of solids and fluids*. University of California, Los Angeles, 2015.

[12] Jyh-Ming Lien and Nancy M. Amato. Approximate convex decomposition of polyhedra. In *Proceedings of the 2007 ACM Symposium on Solid*

and Physical Modeling, SPM '07, pages 121–131, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-666-0.

[13] Khaled Mamou. Approximate Convex Decomposition for Real-Time Collision Detection. In *Game Programming Gems 8*, pages 202–210. Course Technology Press, 2010.

[14] Rodrigo Minetto, Neri Volpato, Jorge Stolfi, Rodrigo MMH Gregori, and Murilo VG da Silva. An optimal algorithm for 3d triangle mesh slicing.

[15] Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions. *ACM Transactions on Graphics*, 32(4):115:1–115:10, 2013.

[16] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *SIGGRAPH 99 Proceedings*, pages 137–146, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. ISBN 0-201-48560-5.

[17] James F. O'Brien and Eric G. Parker. Real-time deformation and fracture in a game environment. In *SCA 09 Proceedings*, pages 165–175, New York, NY, USA, 2009. ACM Press/Addison-Wesley Publishing Co.

[18] James F. O'Brien, Jessica K. Hodgins, and Adam W. Bargteil. Graphical modeling and animation of ductile fracture. In *SIGGRAPH 02 Conference Proceedings*, pages 291–294, New York, NY, USA, 2002. ACM Press/Addison-Wesley Publishing Co.

[19] Jacob Olsen. Realtime procedural terrain generation. 2004.

[20] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. A material point method for snow simulation. *ACM Transactions on Graphics*, 32(4):102, 2013.

[21] Dong-Ming Yan, Wenping Wang, Bruno Lévy, and Yang Liu. Efficient Computation of 3D Clipped Voronoi diagram. In *GMP*, volume 10, pages 269–282. Springer, 2010.

A. User guide

This part of the documentation is focused on the end user, in this case, player. We introduce controls of the application and detail the possibility to adjust the game world.

A.1 Installation guide

This section documents the process of compiling and running the application. We focus on the installation process on *Ubuntu 17.04*, but the code is written in as platform independent and should work on any platform where the library dependencies work.

The packages with library dependencies are available for multiple *Linux* distributions and can be manually compiled and installed on most other operating systems.

Dependencies Following packages are required to compile the application. We also provide versions of packages used to create and test our implementation.

package	version
libbullet-dev	2.83.7+dfsg-5
libirrlicht-dev	1.8.4+dfsg1-1
libcgal-dev	4.9-1build2
voro++-dev	0.4.6+dfsg1-2
HACD	bundled with the application[1]

A C++ compiler capable of compiling the C++14 standard is also needed, we have used GCC version 6.3.0-2ubuntu1 with compilation flags `-std=c++14 -O2 -frounding-math`.

Compiling the application

We assume that all the correct versions of the packages from the previous section are installed. This installation process is automated with the use of *make*. The HADC library is bundled with our code with its own `Makefile`.

The bundled HACD library needs to be compiled first, and then the main application is compiled. The compiling process may take a few minutes. After extracting the archive, the code is compiled by changing into the directory *DestructionInGame* and running `make`.

Key	Function
W/S	pitch down/up
A/D	roll counter-clockwise/clockwise
Q/E	yaw left/right
Z/X	speed up/slow down
Space Bar	shoot or unpause
P	pause the game
ESC	quit the game

Table A.1: Key bindings available for the user.

Hardware requirements

The software was tested on a system with Intel Core i3 550 CPU (3.20GHz × 4), 8GB of DDR3 RAM and 1GB ATI Radeon HD 5770 GPU. Minimal requirements should be lower, but for observing the behavior of the algorithm, a CPU capable of running at least 3 computation threads simultaneously should be used. We also recommend at least 4GB of RAM [2]. The graphics card has to support OpenGL.

Using the application

The application is run by executing the *build/game* file. One optional console parameter is provided. It is `-d`, and it enables debugging information to be rendered. In other words, it draws the collision shapes and bounding boxes around the objects.

The user is able to control a flying vehicle and shoot at the destructible objects. The application accepts the keyboard input. The bindings are listed in table A.1.

A.2 Directory structure

Our application is distributed among multiple folder and files. Here we describe the directory structure and mention the files in them.

src/ is a folder with our source code, both header files and cpp files can be found here.

[2] The memory requirement is not needed for running the application — compilation of CGAL headers with any optimizations enabled require more than 2GB of RAM

media/ directory contains 3D models, textures and *world.cfg* file (see appendix A.3). The other .cfg files are configurations used for experiments. We can also find *cube_X.obj* files, those are cubes with X triangular faces.

include/ folder contains third party header files that are needed for compilation. Only headers associated with HACD are currently in the folder.

lib/ contains the source codes of HACD library in sub-folder hacd/. It is also a target for compiling HACD.

build/ is a target directory for make. After compilation object files and executable file *game* are located here.

A.3 Input data

To allow the user to easily change the game world without the need of recompilation the input text file describing the game setting is provided. The mentioned file is located in `media/world.cfg`.

`world.cfg` format

We show that the structure of the data is straightforward and therefore we have chosen not to use XML or any other standardized format. The file is interpreted by lines. Each line represents one in-game entity. First three lines have special meaning, and the rest are destructible objects. Inside the line, we use semicolons as a separator between data items.

The *first line* describes the sky-box, it is formed of 6 images in given order: top, bottom, left, right, front, back; relative to the starting viewpoint.

All other lines use same nine data items. Those are object model, texture, position (3 coordinates), scale (3 numbers, one for every axis) and mass. Every number is used as a floating point number. The second line describes the ground. Currently, the ground is generated as a solid cube with 1x1x1 size. Therefore model file is not needed. The third line is a description of our controlled vehicle. Beginning with this line, the texture file is an optional parameter and can be left blank. The described format can be seen in fig. A.1.

```
top.jpg;bottom.jpg;left.jpg;right.jpg;front.jpg;back.jpg
;grid.jpg;0;-15;0;1000;10;1000;0
fighter.3ds;;0;20;0;0.5;0.5;0.5;1
building.obj;;-20;-11;-30;2;2;2;500
pyramid.obj;;50;-11;-100;5;10;5;800
...
```

Figure A.1: Example of the *world.cgf*. The first line describes skybox textures, the rest consists of: object; texture; position (x;y;z); scale (x;y;z); mass. The zero mass means that the object is static and does not move.

Object models and textures

The input formats are limited to the support of *Irrlicht*. The detailed list of supported object and texture formats can be found on *Irrlicht* web page [3]. When using a new model file, the mesh needs to be checked for non-manifold geometry to ensure the compatibility with the `CGAL::Nef_polyhedron_3` (appendix B.2).

[3] `http://irrlicht.sourceforge.net/?page_id=45#supportedformats`

B. Implementation internals

This section provides insights on the code from a programmer's point of view. We do not focus on algorithms here, as we already did that in chapter 3. The main focus here is on data representation and division of code into multiple modules.

B.1 Design and Libraries

Logically, the program consists of four components, and they are Physics, Graphics, Mesh Manipulation and Controllers. Controllers connect the other three components together, see fig. 3.2. In the following list, we explain the purpose and summarise used technology in each of them.

Physics provides us with the simulation of our objects and collision detection. We use *Bullet physics* to implement this component (more on physics engines in section 2.1).

Graphics is used for rendering the current state of the physics world. Although it is possible directly use those low-level libraries, we use a library with more abstraction in provided API. Using higher-level libraries means we can save code, but we lose a direct control of rendering process. This thesis is not focused on graphical output and only uses it as a tool for building our application.

We decided to use **Irrlicht Engine** [1]. Irrlicht is a light-weight cross-platform, high-performance engine capable of using either *OpelnGL* or *DirectX*. Our use of Irrlicht is based on using its graphical window as an output of our application and also using Irrlichts event receiver for reading user input.

Controllers ask the Physics to step the simulation and provided collision data and give the information about the current state of the world to the Graphics. The Controllers also read the user input which is done by parts of *Irrlicht engine*. Mesh Manipulation component is called to create new objects and react to the collision provided by the Physics.

Mesh Manipulation provides tools for creating objects, creating a Voronoi cell (done by *Voro++*, converting data and handling the collisions. To be able to do the mentioned tasks, Mesh Manipulation works closely

[1] `http://irrlicht.sourceforge.net/`

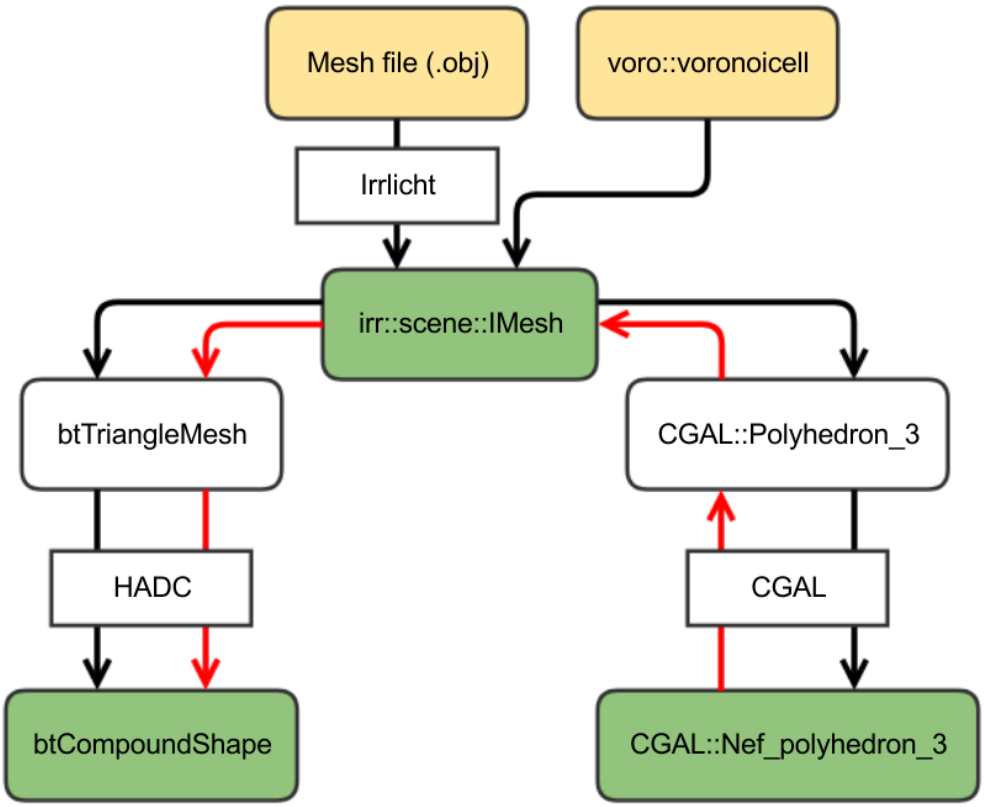

Figure B.1: Data conversion diagram. Input formats - yellow, required formats - green. Black lines show used conversions, red lines show required process after a change of the shape of an object. Rectangles signify the library used to make the conversion, if none a member function of `gg::MMeshManipulators` is used.

with *CGAL* library and also use *HACD* library for convex decomposition.

B.2 Data representation

As a result of using multiple libraries for a multitude of tasks, we have to deal with a lot of different data representations of the same objects.

We introduce the data structures that are used to represent one in-game object for multiple purposes. We can see how different data types are converted in fig. B.1.

`btRigidBody`

Bullet physics uses this class to hold information about a rigid collision object. For us, most important part of the body is a collision shape.

The collision shape can be of multiple types, most notably a convex hull or a primitive geometric shape, a triangular mesh or a compound shape. We need the collision shape to be as close to a visual mesh as possible. Because calculating collision between triangular meshes is not implemented in the *Bullet physics* and would be too costly even if implemented, we choose the representation by compound shape.

One more parameter of `btRigidBody` to consider is the object mass. Bodies with mass set to zero (or negative value) are considered static objects and do not move. Bodies with positive mass react to gravity and other external forces, their center of gravity is set to their respective origin of local coordinates. This poses a problem when the origin is not inside the object.

We try to solve the problem of displaced origin by calculating a center of the bounding box of the object and then translating all vertices in a way that the center becomes an origin. After the origin is changed, we must appropriately adjust the position of the object. We deployed this approach to newly created objects. The difference between where the real centre of gravity should be, and where the center of gravity used by Bullet physics is, is not visible. However, we failed to solve this problem with objects with their mesh changed. This results in the visibly displaced center of gravity and occasional wrong position of newly created objects. Because this problem has no impact on the simulation of the destructible environment, we decided against trying to solve it.

`irr::scene::ISceneNode`

Graphical object in the *Irrlicht engine* is represented by this class. It is an abstract class instantiated into multiple types of graphical objects, e. g. lights, cameras, animations, particle systems. To represent our objects, we are using `irr::scene::IMeshSceneNode`. `irr::scene::IMesh` is the data structure inside `irr::scene::IMeshSceneNode` that stores the mesh information.

`irr::scene::IMesh`

This class stores the mesh information in multiple mesh buffers. Each buffer has an array of vertices and an array of indices. Every index in the array of indices refers to one vertex. However, we found out that not every vertex

is referred to, and therefore valid. Indices divided into consecutive non-intersecting triples form a triangular face of a mesh.

`CGAL::Nef_polyhedron_3`

"A Nef-poly-he-dron in dimension d is a point set $P \subseteq \mathbb{R}^d$ generated from a finite number of open halfspaces by set complement and set intersection operations." [2] In other words, `CGAL::Nef_polyhedron_3` is a boundary represented data structure closed under Boolean operations. This make it ideal to perform our boolean operation on. This structure imposes a restriction on the data we can use to represent our objects, the underlying halfedge data structure [3] is restricted to orientable 2-manifolds. Common examples of non-manifold geometry: open geometry, two neighbouring faces with opposite normals, two faces sharing vertex but no edge, self-intersecting geometry, inside faces.

`gg::MObject`

`gg::MObject` is a class designed to unite all data about one in-game object into one structure. It includes `irr::scene::ISceneNode`, `btRigidBody` and `CGAL::Nef_polyhedron_3`. It implements the mechanism ensuring that upon deletion of an object, its parts are first removed from their respective engines, and then safely deallocated before `gg::MObject` itself is destroyed.

B.3 Modules

In this section, we describe the functionality of each program module. Third party software is not described here, details about used libraries can be found in chapter 2. Interactions between modules are visualized in fig. B.2. *Irrlicht Engine* is not present in the diagram because we do not structurally depend on it. Every module is contained in a file of the same name, as a class with that name prefixed with letter M (i. e. Object Creator can be found in file ObjectCreator.cpp and is implemented as class `gg::MObjectCreator`). Also, namespace `gg` identifies exact components of the application implemented for this thesis.

[2] `http://doc.cgal.org/latest/Nef_3/index.html#Nef_3Definition`

[3] `doc.cgal.org/latest/HalfedgeDS/index.html#Chapter_Halfedge_Data_Structures`

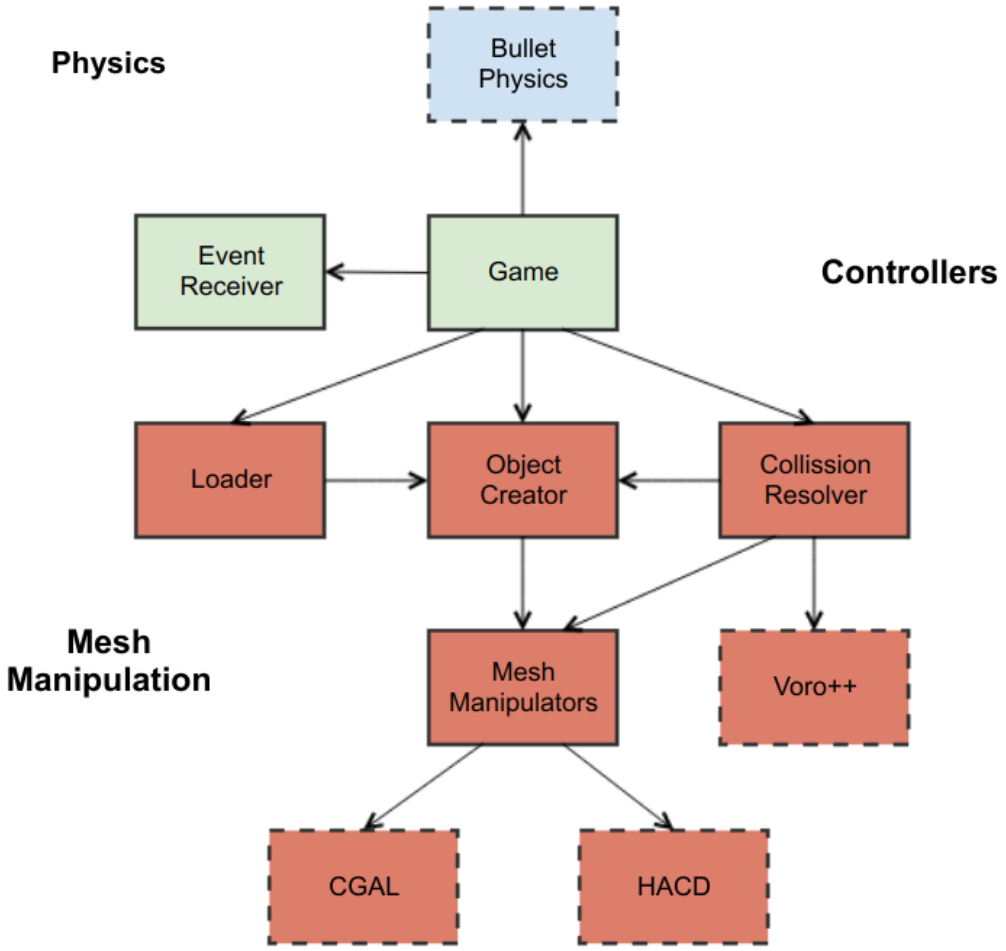

Figure B.2: Software architecture shown on diagram of relationships of program modules. Third party software is highlighted in dashed rectangles.

Game module holds `gg::MGame` class which is the core of the application. The communication with the physics engine, the graphical engine and mesh manipulation parts of the software is managed from here.

Event Receiver module implements the instance of `irr::IEventReceiver` from Irrlicht engine and it is used to read the user input.

Loader is only used for initializing the application. It parses the data that describe the game environment from the *medial/world.cfg* file, constructs the objects using `gg::MObjectCreator`, and returns the set of constructed objects.

Object Creator implements the `gg::MObjectCreator` class designed in a Builder pattern to provide initialization for the data contained inside `gg::MObject`.

There are three member functions that allow us to create `gg::MObject`s with different behaviours from the same set of input parameters (see appendix A.3). We can create a destructible object, an indestructible object with box collision shape and a rectangular indestructible object without input mesh. Those functions are meant to be used exclusively for application initialization, as they generate a new convex decomposition and `CGAL::Nef_polyhedron_3` from their mesh.

Two more kinds of objects can be created: a projectile that is shot from the given position with the given impulse and a destructible object with temporary collision shape (sphere shaped). Because the object with temporary collision shape is constructed while the game is being played and construction of `CGAL::Nef_polyhedron_3` can take a longer time, to gain control over the construction, we will construct it beforehand and then provide it to the Object Creator.

Collision Resolver implements the Collision handling process, described in section 3.3. Conversions between data formats that are required during the process are externalized to Mesh Manipulators module to maintain the code readable.

Mesh Manipulators provides set of utility functions. Because different libraries are used for physics simulation, rendering and geometric manipulation, those functions provide means for converting data between different formats.