nice journal name :)

# Towards real-time physics in a 2D survival-based game

Björn Aheimer, Felix Röhr, Daniel Six, Aleksis Vezenkov, Anietta Weckauff

you can skip FA in the abstract (if you want)

**Abstract**—Surviving Sarntal is an action-packed arcade endless runner. The player has to navigate a hiker through demanding terrain all whilst dodging dangerous rockfall. Although the idea and a rough initial implementation of the game were devised in the Ferienakademie 2023, we rewrote the entire game from scratch in C++ using only a small number of libraries. Goal of the project was the design of a finished game following good software engineering practice with particular focus on creating a performant physics engine for rigid body simulation. This report explains our development approach, the game logic, and it discusses our findings in all stages from research, development to deployment.

**Index Terms**—Game Development, Hermite Splines, Rigid Body Simulation, Physics Engine, SAT, Collision Detection and Resolution, C++

keywords?

add [2] in 2.1 mention it is open-source etc. Add a license if not there yet.

font inconsistent with the rest

## 1 Introduction

The objective of the project „Towards real-time physics in a 2D survival-based game“ is a comprehensive restructuring and ipprovement of the original game „Surviving Sarntal“. This comprises three major work packages: Software engineering, development of real-time physics and game engineering.

The first part of the project focuses on improving the code by adherence to software engineering principles. This includes comprehensive refactoring of the original code base, designing a new and maintainable software architecture, establishing protocols for development, testing and clean code as well as creating a platform for detailed documentation for users and developers. The project's continuous progress was ensured by applying Scrum principles and structuring it in sprints [1]. goal

The primary ~~objective~~ of the development phase is the enhancement of the real-time physics, which involves the implementation of realistic 2D physics without compromising real-time performance. This was accomplished by decoupling the physics simulation rate from the frame rate. Furthermore, the project includes the development and implementation of improved models and algorithms for interaction between game elements such as collision detection and handling.

Moreover, the project contributes to the enhancement of the game design as a whole. A number of new elements have been implemented, such as configurable terrain generation, new items with improved functionality, new rock generation, increasing difficulty and updated graphics.

In essence, the project was developed in accordance with the vision of creating an enhanced version of the game „Surviving Sarntal“ that would offer users a realistic physics simulation, engaging gameplay, and aesthetically pleasing visuals as well as creating a maintainable code base ~~[2]~~.

• *The authors are in the B.Sc. Informatik at the University of Stuttgart*
*Examiner: Prof. Dr. Miriam Schulte, Univ. of Stgt., IPVS/SGS*
*Supervisor: Carme Homs-Pons, M.Sc., Univ. of Stgt., IPVS/SGS*

### 1.1 The Game

The general goal of the game is for the Hiker to survive as long as possible and climb as high as possible. ~~A~~ player ~~must do that while~~ is being chased my a monster and evading ~~the~~ the rocks that roll down the mountain. Moreover, the player may collect items that give the hiker additional benefits like restoring health and adding a protective shield. A comprehensive guid to the controls can be found in the project documentation [3] and at the start of the game under the menu "Instructions".

### 1.2 Transformation of the original game

The original game „Surviving Sarntal“ was created within a few days at the „Ferienakademie 2023“. In the course of this project, we developed a new version of the game from scratch. Therefore, our version of Surviving Sarntal and the original version differ in various ways.

is it really from scratch?

#### 1.2.1 ECS vs. OOP

The original game was based on the Entity-Component-System (ECS) using Flecs [4]. As a team, we decided to alter the project's fundamental architecture from relying on the ECS to an object-oriented programming (OOP) approach. Given our familiarity with OOP, we came to the conclusion that transitioning from ECS to OOP would allow us to create a code base that could be easily comprehended, expanded and maintained by our team.

#### 1.2.2 Transformation of Graphics

During the development process of the new version of the game, we decided that a complete reworking of the game's graphics was necessary. New and creative graphical elements were created for all aspects of the game, including the entities, terrain, and background, and were integrated into the rendering phase of the game.

You are using past tense

Again 1.2.X are in past tense
↳ I'd use present for consistently

I want you to mention all the figures you include in the text at least once.

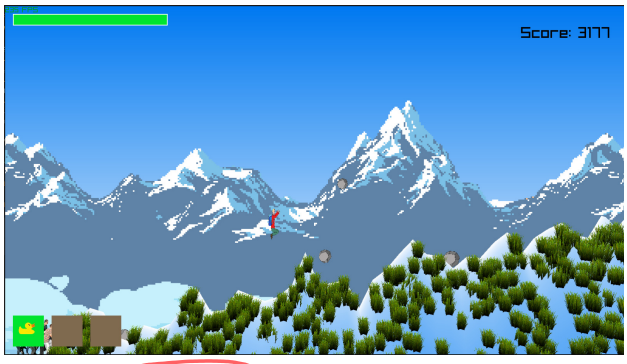



Figure 1: Screenhot of the original game graphics.

? ← heiß!

Figure 2: Screenhot of the renewed game graphics.

fig 1 & 2 could be merge together

+ basics users have all info at reach

#### 1.2.3 Addition of a game menu

As a new feature, which is intended to enhance the overall user experience of the game, we decided to included a menu. Once the game has been initiated, the user is presented with a start menu, which allows the user to access instructions, modify settings or begin the gameplay. After the end of the game, the player is redirected to an end screen that enables them to start another round of gameplay. When pausing the game, the player is redirected to a pause screen that gives the player the opportunity to either continue gameplay or go back to the main menu.

#### 1.2.4 Modular Architecture

As part of the software engineering part of the project we designed a modular game architecture that provides several key benefits:

- **Ease of Maintenance:** Since components such as rendering, physics, and spawning are isolated from each other, it is easier to modify, debug, or upgrade parts of the game without introducing side effects.
- **Maintainability:** New features or game elements (e.g., new items, enemies, or gameplay mechanics) can be added by extending existing systems without requiring significant changes to the core architecture.
- **Testing and Debugging:** The `DevMode` enables specific components to be tested in isolation, streamlining the debugging process and allowing developers to focus on individual elements of gameplay.

New terrain missing!

not the same?

This modular structure enables the development team to build complex systems in a more controlled and efficient manner, leading to a highly extensible and maintainable game.

## 2 Getting Started

The project is compatible with Ubuntu 24.04.1 and also supports macOS. It includes several bash scripts to simplify installation, running, and testing. A platform-agnostic script handles installing dependencies across different operating systems. There are separate installation processes for playing the game and for development, with an additional script provided for developers. More information on contributing to development can be found in the project documentation. [3].

### 2.1 ~~Installation~~ Installing & Running

In the event that only running the game is desired, the script `user_setup.sh` [2] can be used to facilitate the installation of the required packages and dependencies. In that case, it is only necessary to run the following command from the root project directory:

```
$ source build-utils/user_setup.sh
```

The following packages will be installed based on the operating system: The installation of the packages on MacOS requires the use of Homebrew. If you do not have it installed you can still run the `user_setup.sh` script, which will install it.

- **Ubuntu:** `build-essential cmake xorg-dev libsdl2-dev`
- **MacOS:** `cmake sdl2`.

### ~~2.2 Running the Game~~

Once all of the dependencies have been installed, the script `run.sh` can be executed in order to run the game:

```
$ source build-utils/run.sh
```

### 2.3 YAML Configuration

decide for or against Oxford comma and make it consistent

In this project, a YAML configuration file is used to manage settings and parameters that control various aspects of the game, such as behavior, resources, and environmental constants. The human-readable YAML format allows easy modification of game elements, without the need for recompilation. This approach enhances development workflow by supporting rapid iterations and testing. The configuration includes:

- **Run Mode and Game Settings:** The configuration begins with a global setting for development mode (`run-dev-mode`). When enabled, it provides additional functionality for developers, helping to debug specific components.
- **Assets:** Specifies all audio and visual assets for the game.
- **Items:** Configures game items with their attributes.
- **Landmarks:** Defines landmarks, such as the Statue of Liberty or Mount Everest, which act as height markers during gameplay.

references?

- **Game Constants:** Includes parameters for in-game mechanics like player movement, health, and environmental interactions. Constants for terrain, item spawning, and physics can be adjusted here, allowing developers to fine-tune gameplay by modifying values such as jump velocity or rock spawn frequency.

### 2.4 Development Mode

A dedicated `DevMode` is provided, allowing developers to test the game in a controlled environment. This mode bypasses the standard game flow to aid in debugging and fine-tuning specific mechanics.

## 3 Game Architecture

The game is built using a highly modular architecture that allows for flexibility, scalability, and ease of maintenance. The modular design breaks down the game into distinct components, each responsible for specific functionality as seen in the architecture diagram 3. This approach ensures that different parts of the game can be developed, tested, and modified independently, making the overall system more manageable. Below, we describe the general architecture and how different parts contribute to building the game.

add link to figure

### ~~3.1 Core Components and Services~~

The game is initialized through a `GameFactory` class, which is responsible for building and configuring all core components and dependencies. These components can be categorized into services, game entities, menu systems, rendering pipelines, spawning systems, and the physics engine.

#### *3.1.1 Services*

Make this 3.1

Several core services are initiated within the `GameFactory` to manage resources, input, and audio:

- **ConfigService:** A singleton pattern is used to handle game configuration (including YAML settings), ensuring game constants and configurations are readily available throughout the game.
- **InputService:** A singleton service responsible for managing input from various devices, such as gamepads and keyboards.
- **ResourceService:** Handles loading and management of game assets such as textures, audio, and other resources. It depends on the `ConfigManager` to load the assets specified in the YAML file.
- **AudioService:** Manages sound effects and background music. It interacts with the `ResourceManager` to load sound files and handle audio playback within the game.
- **DifficultyService:** A singleton service that manages the difficulty progression during the gameplay.

### 3.2 Game Entities

Key in-game entities, such as the terrain, hiker, and inventory, are created and managed by the `GameFactory`. These entities interact with the world and with each other, providing the core gameplay functionality.

- **Terrain:** Generated based on game constants and managed by the `Terrain` class. It determines the physical environment in which the player interacts.
- **Hiker:** The main player character, represented by the `Hiker` class.
- **Monster:** An antagonist entity represented by the `Monster` class, which adds an element of challenge for the player, acting as a threat.
- **Inventory:** Manages the player's items and inventory.

These entities are bound together within the `World` class, which encapsulates the terrain, hiker, monster, and inventory, managing their interactions.

### 3.3 Spawner System

The game includes a system for dynamically spawning items and rocks during gameplay:

- **ItemSpawner:** Spawns items in the world based on configuration from the YAML file (e.g., item spawn rates and types).
- **RockSpawner:** Dynamically spawns rocks of various sizes, speeds, and densities during gameplay, adding an element of challenge.
- **Spawner:** A unified spawner that manages both the `ItemSpawner` and `RockSpawner`. It also initiates the terrain generation.

This modular spawner system allows for dynamic game elements that can evolve based on player progress or difficulty.

### 3.4 Physics and Collision Detection

The game incorporates a robust physics engine to handle movement, collisions, and interactions between entities:

- **Accelerator:** Applies acceleration forces to the entities, such as gravity and other forces affecting the hiker or objects in the game world.
- **CollisionDetector:** Detects collisions between entities (e.g., hiker and terrain, rocks, or inventory items) to trigger corresponding responses.
- **CollisionHandler:** Handles the game logic triggered by collisions, such as taking damage or picking up items.
- **Destructor:** Handles the accurate and timely destruction of entities.
- **Positioner:** Handles the correct positioning of the entities within the game
- **PhysicsEngine:** The central physics engine that manages all other physics modules.

By abstracting collision detection and handling into separate classes, the game ensures that its physics are flexible and can easily be adapted or expanded upon.

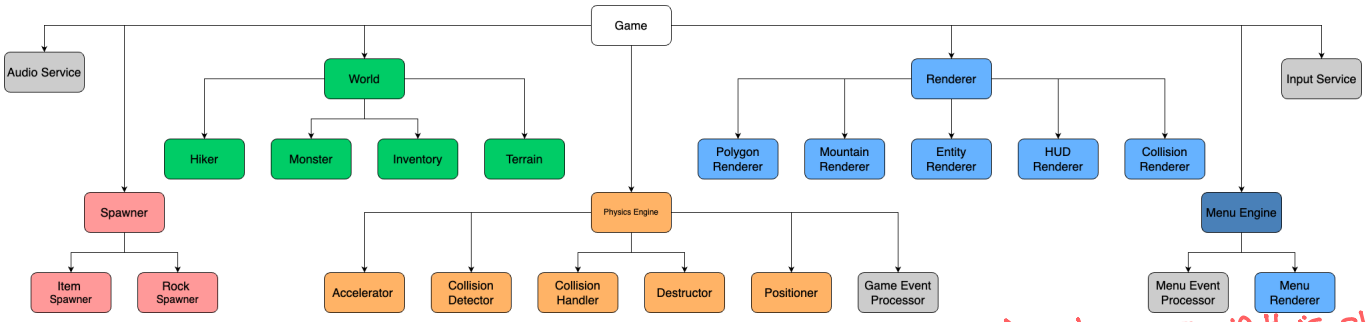



Figure 3: Architectural Diagram

1) not sure if this should be a standalone function
2) placed before architecture?

### 3.5 Menus and UI

The game uses a sophisticated menu system to handle the user interface, including in-game menus. This system is composed of:

- **MenuEngine:** Responsible for managing the game's various menus, including the start screen, settings, and pause menus. It also handles user input in these menus.
- **FullMenuRenderer:** Renders the menu components on the screen.
- **MenuEventProcessor:** Handles events that occur within the menus, such as button clicks and navigation.

This system ensures that the menu and UI are distinct from the core game logic, allowing independent development and modification of the user interface.

### 3.6 Renderer

The game rendering is divided into several subsystems to ensure modular and efficient drawing of different elements:

- **PolygonRenderer:** Used for drawing polygonal shapes, typically for terrain or other geometric elements.
- **MountainRenderer:** A specialized renderer for drawing the mountain and environmental elements in the background.
- **EntityRenderer:** Handles rendering of game entities (e.g., the hiker, inventory items, and the monster).
- **HUDRenderer:** Renders the HUD, displaying information such as health, score, and inventory status.
- **Renderer:** The central renderer that coordinates all these subsystems. It integrates the `World` entities and renders the terrain, entities, and HUD together.

This modular rendering system separates concerns, ensuring that changes to how entities are drawn, or the addition of new visual elements, can be done without affecting the rest of the game.

### 3.7 Development Mode

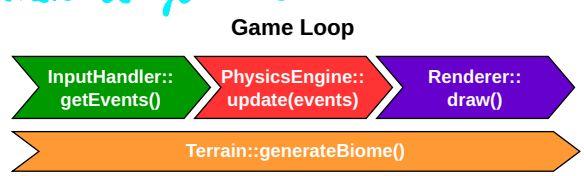
repetition!

A development mode (`DevMode`) is provided, which allows developers to test the game in a controlled environment. This mode bypasses the standard game flow to facilitate debugging and fine-tuning of specific mechanics.

## 4 The Game Loop

The game loop is the central control unit of our program. It contains all the logic and data needed to run the actual gameplay. Our game loop is depicted in Figure 4. For the most part, it adheres to the industry standard as described by Nystrom [5] and the Unity framework [6]. Hence, we will only briefly describe its functionality here and then focus on the underlying logic of the physics engine and the renderer.

where do you do that?




✓ very nice :)

Figure 4: The game loop represents the fundamental structure of the game and is comprised of three distinct phases. The terrain generation is performed concurrently.

At the start of the game loop, input from the player, be it via keyboard or gamepad, is registered and passed to the physics engine. The physics engine progresses the simulation in one or more constant time steps between frames (c.f. decoupling in section 6.3), thereby processing the collected input events. Finally, the renderer draws the new frame representing the updated world state.

Due to the sequential nature of input receipt, input processing, simulating a physics step, and rendering the resulting new state of the world, the game loop is also inherently sequential. For very large games or powerful frameworks such as the aforementioned Unity framework, a parallelization becomes inevitable, but in our game, this structure suffices. As described in section 6.9, the physics engine does not have to run on a separate thread for our purpose. Merely the terrain generation is extracted from the spawning phase of the physics engine in order to guarantee a smooth transition between biomes during gameplay.

make it a link

## 5 The Terrain

The terrain that the hiker walks on is randomly generated for every game. It is represented by 2D-Splines and is approximated by polylines for collision detection and rendering.

### 5.1 Structure

Internally, the terrain consists of a list of biomes. Each biome in turn ~~has a ground that~~ consists of a list of points $(p_0, p_2, \ldots, p_n)$. For a reasonably smooth terrain, we decided to use 2D $C^1$-continuous Hermite-Splines to interpolate in between the points. Here, every spline interpolates between a start $p_i = (x_i, y_i)^T$ and end point $p_{i+1} = (x_{i+1}, y_{i+1})^T$ given in 2D space. To do this, we use separate 1D $C^1$-continuous hermite splines $H_x(t), H_y(t)$ for $x$- and $y$-Direction that are both dependent on an artificial third prameter $t$. This parameter $t$ is roughly equivalent to the distance along the terrain. Therefore, it is set to $t_0 = 0$ for the first point and then calculated using euclidian distance for every further point.

To get the position of the terrain at a given $t$ we combine the two spline evaluations into a point $(H_x(t), H_y(t))^T$. We can calculate the $C^1$-continuous Hermite-Spline $H_{i,x}(t)$ between a given start and end point $x_i, x_{i+1}$ using start and end derivatives $dx_i, dx_{i+1}$ and a start value $t_i$ for $t$ as follows. The procedure is analogous for $H_{i,y}(t)$.

$$
\begin{aligned}
t_{i+1} =& t_i + \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \\
H_{i,x}(t) =& (1 - 3t^2 + 2t^3)x_i + (3t^2 - 2t^3)x_{i+1} \\
&+ (t - 2t^2 + t^3)(x_{i+1} - x_i)dx_i \\
&+ (t^3 - t^2)(x_{i+1} - x_i)dx_{i+1}
\end{aligned}
$$

Feels in the middle, move to end of 5.1

A good overview over hermite splines can be found in [7].

We also experimented with $C^2$-continuous splines. In contrast to our method, here the first derivatives at the start and end of a spline are determined by conditions for the second order derivative at the start and end and do not need to be provided. However, these splines typically require us to know all interpolated points at once. Therefore, this approach is unsuitable for our generation that is performed in an online manner. For this reason, we chose to stick with $C^1$-continuous splines and provide the first order derivatives.

good that you include it

These first order derivatives can be freely chosen but need to match at the borders between splines since the splines are $C^1$-continuous. Therefore, we provided the start derivative $(dx_0, dy_0)^T = (1, 0)^T$ that signifies even terrain. From there, we set the derivative of the end of a spline piece to the slope we would get for piecewise linear interpolation:

that corresponds to?

$$
\begin{aligned}
&\forall i \in \{0, 1, \ldots, n-1\} : \\
&\quad (dx_{i+1}, dy_{i+1}) = (x_{i+1} - x_i, y_{i+1} - y_i)
\end{aligned}
$$

The individual spline pieces $H_i(t) = (H_{i,x}(t), H_{i,y}(t))^T$ are then combined into one continuous function $H(t)$, that allows us to interpolate the terrain at any $t$-value between the first and last point. Using this method, we can generate a spline-based interpolation of the ground points in an online setting. This is particularly useful for the generation of the terrain, where the points are generated iteratively using the existing spline representation.

### 5.2 Generation

At the start of the game, the terrain is generated with one start biome of fixed length. After that, a new biome of the same fixed length is generated whenever the hiker gets close to the right border of the currently generated terrain. This is done concurrently to not disturb the playing experience. Similarily, whenever a biome is no longer visible it is removed from the terrrain.

The basepoints used to specify the ground of each biome are generated in an iterative manner. The overall form of the ground in a biome is specified by the terrain phases $ph_1, ph_2, \ldots, ph_n$ of this biome as well as a special starting terrain phase $ph_s$ that determines the first few point of the ground. Here, each terrain phase has the following format:

| | |
|---|---|
| $ph = (\delta, r, m)$ | where: |
| $\delta \in \mathbb{R}^2$ : | Average vector to be added to the last point to generate the next one |
| $r \in [0, 1)$ : | Randomness, specifies how much we can rotate the average vector to generate the next point |
| $m \in \mathbb{N}$ : | Number of points to be generated using this phase before selecting a new phase |

To generate a new point $p_{i+1}$ using the existing terrain up to point $p_i$ and a terrain phase $ph = (\delta, r, m)$, we first determine the angular range $[min, max]$ in which generation of a new point is allowed using the Algorithm 1 for $max$ and its anologon for $min$. Then, we generate a new point by adding $\delta$ rotated to a random angle in the range $[min, max]$. If the anglular range $[min, max]$ is empty, no such point can be generated. Therefore, we need to retrace some of the already generated points and try to generate the terrain starting from an earlier point.

#### 5.2.1 *Calculating the anglular range*

The angular range in which generation is allowed is defined by a $min$ and $max$ angle which are determined using a bisection method. More precisely, we iteratively determine these angles by trying different angles and checking wheter the terrain would fulfill the constraints if the new point was generated using the current angle and terrain phase. If so, we can try a larger angle in case of $max$ and a smaller angle in case of $min$. The delta angle $\delta_a$ in beween tried angles is halfed for every successful try. If $\delta_a$ gets small enough or we exceed a fixed limit of tries, we return the best previously found angle if one exists or that there is no suitable angle. This procedure is described in detail for determining $max$ in Algorithm 1. The process is analogous for determining $min$. The process of determining $min$ and $max$ in an example case is illustrated in Figure 5.

Add links

#### 5.2.2 *Retracing*

Even though we check the constraints for every generated point, it can happen that we cannot generate a new point from the existing terrain. In that case, we need to retrace some of the previously generated terrain. If this happens for the first time, we remove a fixed number of points from the terrain. After that, the generation continues as normal. This procedure is illustrated in Figure 6.

If we cannot regenerate all points that were removed before we retrace again, then we increment how many

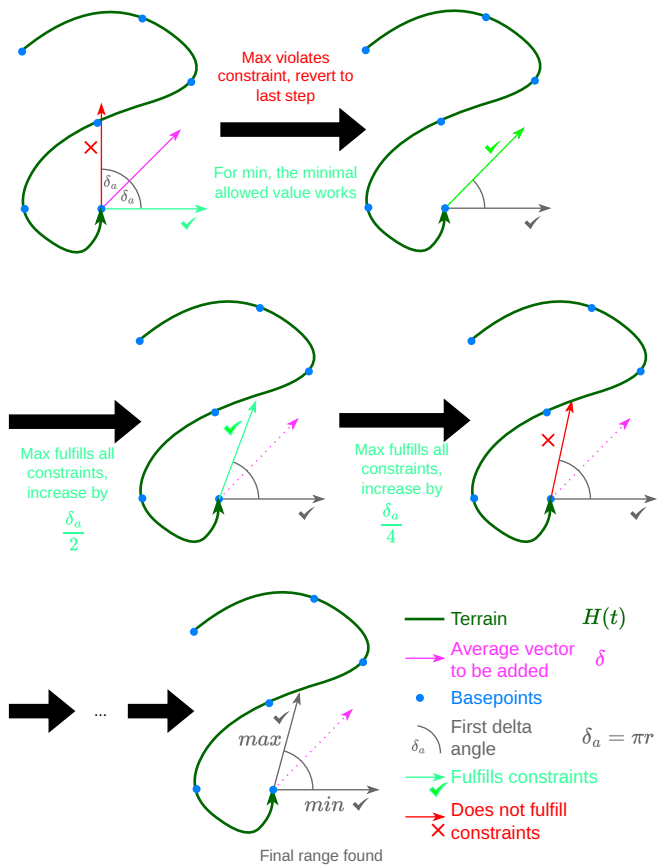



Figure 5: Calculating the range $[min, max]$ of angles in which the generation of a new point from the last point of the existing terrain and using a given terrain phase is possible.

**Algorithm 1** Calculating the maximum angle

**Input:** $ph = (\delta, r, m) \in \mathbb{R}^2 \times [0, 1) \times \mathbb{N}$,
$p_i \in \mathbb{R}^2$, $terrain \in (\mathbb{R}^2)^n$
**Output:** $angle \in \{\perp\} \cup [-\pi r, \pi r)$

**procedure** CALCULATEMAXANGLE($ph, p_i, terrain$)
  $c_{max} \leftarrow 30$; $minStep \leftarrow 0.01$; $c \leftarrow 0$
  $min_a \leftarrow -\pi r$; $max_a \leftarrow \pi r$; $\delta_a \leftarrow max_a$
  $step \leftarrow max_a$; $angle \leftarrow max_a$
  **while** $(c < c_{max} \wedge step > minStep$
    $\wedge angle \leq max_a)$ **do**
    $next \leftarrow p_i + rotate(\delta, angle)$
    **if** $fulfillsConstraints(p_i, next, terrain)$ **then**
      $\delta_a \leftarrow \frac{\delta_a}{2}$ ▷ lower step size
      $angle \leftarrow angle + \delta_a$
    **else**
      $angle \leftarrow angle - \delta_a$ ▷ take back last step
    **end if**
    **if** $angle \leq min_a$ **then**
      ▷ No viable angle in $(min_a, max_a]$
      $angle \leftarrow max_a$
      $\delta_a \leftarrow \frac{\delta_a}{2}$ ▷ Try again with smaller step size
    **end if**
    $c \leftarrow c + 1$
  **end while**
  **if** $c = c_{max}$ **then** ▷ No viable angle found
    **return** $\perp$
  **end if**
  $angle \leftarrow angle - \delta_a$ ▷ Restore last viable angle
  **return** $angle$
**end procedure**

points need to be removed by a fixed number and retrace again. We also tried an exponentially increasing retrace count. However, that regularily lead to the entire ground being regenerated and the generation took longer overall. Therefore, we decided to use the linearly increasing retrace count.

### 5.3 Contraints

To determine the anglular range for generation and also to verify that the newly generated point is appropriate, we formulate several constraints on the terrain. These constraints ensure that the terrain does not intersect with itseld, the problems that that there are no too thin parts in the terrain, that it is navigatable by the hiker and does not generate backwards. ~~What problems~~ would arise in the terrain generation should these constraints be violated can be seen in Figure 7.

### 5.4 Representation

For both rendering and collision detection with rocks and the hiker, the spline-based representation of the terrain is impractical. Therefore, the splines are approximated differently for each purpose:

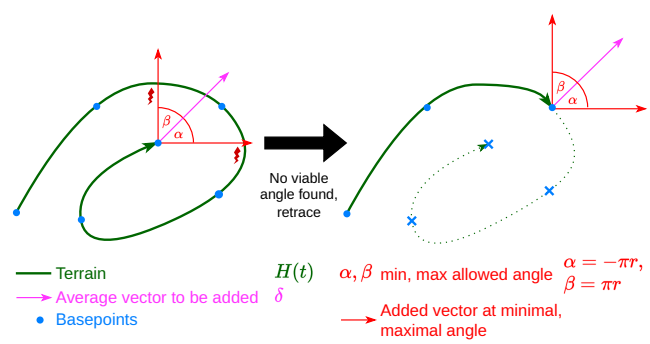



Figure 6: Trying to generate a new point with the current terrain phase $ph = (\delta, r, m)$ when the constraints do not allow that. Results in retracing, that is removing of the last few generated points.

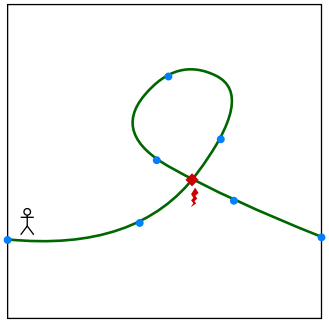
Rendering:

To make the terrain visually appealing, the ground $H(t)$ is approximated by a polyline $T_{rend}$ with a relatively high resolution $res_{rend}$:

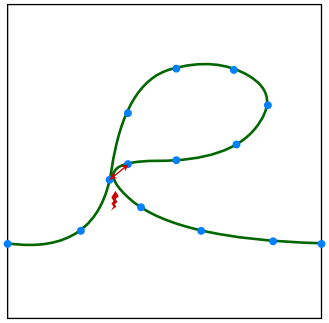
$$T_{rend} = \Big(H(0), H(res_{rend}), H(2 \cdot res_{rend}), \ldots, H\left(\left\lfloor \frac{t_n}{res_{rend}} - 1 \right\rfloor \cdot res_{rend}\right), H(t_n)\Big)$$

Collision

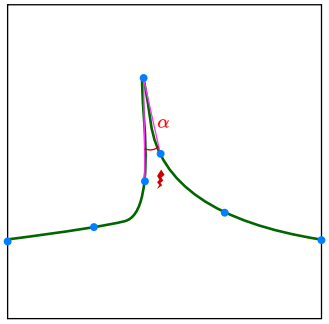
To improve performance, the ground is approximated by a



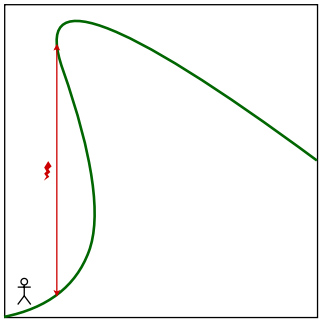
(a) No Selfpenetration



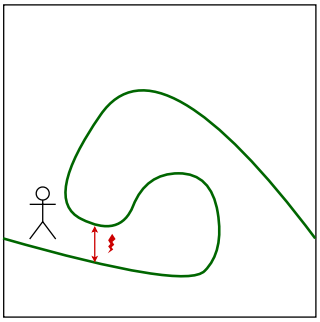
(b) Minimal Basepoint Distance



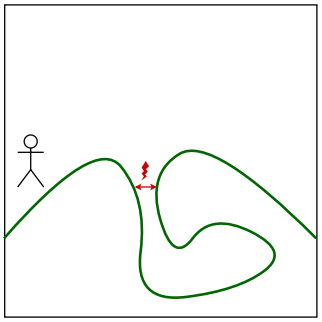

(c) Minimal Basepoint Angle



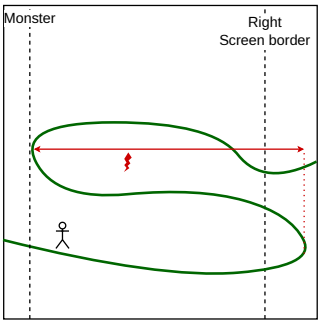
(d) Terrain Jumpeable



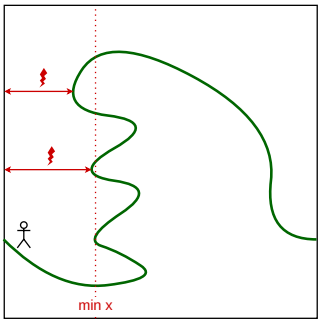
(e) Hiker Clearance (Height)



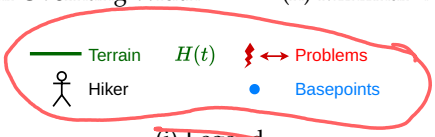
(f) Hiker Clearance (Width)




(g) Maximal Overhang Width




(h) Minimal x-Pos




not sure if needed

~~(i) Legend~~

Figure 7: Examples of terrains that could be generated if the constraints are ignored.

polyline $T_{i,cd}$ for each spline-piece $H_i(t)$ using a coarser resolution $res_{cd}$. Combining all of these piecewise approximations into one results in the representation $T_{cd}$ for collision detection:

$$\forall i \in \{0, 1, \ldots, n-1\} :$$
$$T_{i,cd} = \Big( H(t_i), H(t_i + res_{cd}), H(t_i + 2 \cdot res_{cd}), \ldots, H\left(t_i + \left\lfloor \frac{t_{i+1} - t_i}{res_{cd}} - 1 \right\rfloor \cdot res_{cd}\right), H(t_{i+1}) \Big)$$

This type of segmented representation is particulatrily useful for an efficient preselection of relevant terrain sections in the collision detection (see section 6.6.3).

## 6 Rigid Body Dynamics

At the heart of the game loop lies the physics engine. The following section provides an overview of the model for the game world, the movement equations, techniques used for collision detection and resolution, as well as a discussion of the performance. We also describe problems we encountered and include a comprehensive survey of literature in the area of physics engines for game development.

### 6.1 Decoupling of frame rate and physics rate

As mentioned in section 4, the physics loop is executed one or more times per frame, progressing the simulation in constant time steps $\delta t$. Having the world state progress at constant speed is good practice as this allows the game to run smoothly and continuously regardless of the varying framerate (frames per second, fps).

We base our implementation on the physics loop proposed by Fiedler [8]. It uses a time accumulator which „collects“ the time between the last and the current frame. The time in the accumulator is then consumed by the physics engine in constant time steps $\delta t$ before the next frame is rendered. This ensures that no time is lost and even small rest times that were not simulated between frames are accumulated and processed later.

Choosing the $\delta t$ is no trivial manner (c.f. section 6.9). Because our $\delta t$ is small and the visual benefit is neglectable, we deviate from Fiedler's implementation [8] and refrain from interpolating the time residue before rendering.

### 6.2 The Physics Update

In one time step of the physics engine, updates are executed in the following order (see Fig. 8):

1) Spawning: Items and rocks are spawned, the terrain generation thread is started or joined if necessary.
2) Event processing: User input events are mapped to state changes of the world and the hiker.
3) Accelerator: (Gravitational) Forces are applied to the entities.
4) Positioner: The positions of entities are updated.
5) Collision Handling: Collisions are detected and resolved. This step is performed multiple times to ensure stability of the simulation.
6) Destructor: An entity will be destroyed if it is out of scope or otherwise undergoes a process of destruction.

shorten if needed, as Fig already contains the infos

(*) This is about rock generation right? if yes, 6.3 tittle is not appropiate. Distinction is needed!

I miss a reference on how is the rock type selected. What about velocity distribution?

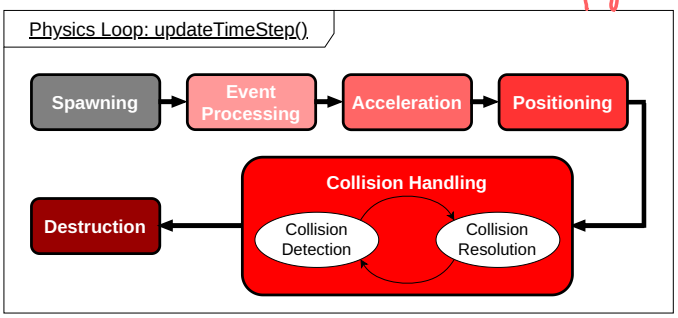



Figure 8: The physics time step, progressing the world state by $\delta t$. Performed one or more times per frame.

### 6.3 Spawning and Destruction

Items are continuously spawned at random positions above the terrain. The frequency of spawning is dependent on a configurable spawn weight. Different types of rocks (e.g. snow/ ice/ lava/ crystal rock) are continuously spawned and propelled through the air. In order to guarantee the randomness of the generated polygons, it is first necessary to generate a random set of points within a specified radius. This is followed by the application of Andrew's monotone chain algorithm [9], which is used to determine the convex hull of the set of points. The convex hull represents the vertices of the generated polygon. New terrain is also generated concurrently once the player approaches the end of the curernt biome.

At the end of a physics update, rocks that have left the scope of the world or hit the hiker are destroyed and removed. Similarly, old biomes are destructed. This garbage collection ensures a consistent performance no matter how long the game runs.

### 6.4 Rock State

In the following, we demonstrate the logic of the physics engine with rocks that are modelled as convex polygons. The monster and hiker are handled analogously with some exceptions that are briefly mentioned afterwards.

Our physics engine is impulse-based as proposed by Baraff [10]. In his paper, he describes how to handle a rigid bodiy simulation in a three dimensional world and how to resolve collisions by applying impulses after they have been detected. However, due to the game being two-dimensional, we were able to reduce Baraff's equations. For instance, the rotation of a two-dimensional object around its center of mass (short: centroid) only has one degree of freedom (DOF) instead of three in the 3D case. ~~In the next section, our adapted equations are presented~~.

In every timestep of the simulation, the dynamic properties of a rock change. We define a polygon's position $\vec{x} \in \mathbb{R}^2$ as the coordinates of its centroid. Its rotation angle $\theta \in \mathbb{R}$ is given in radians. The coordinates of the vertices are stored as body space vertices $\vec{r_i} \in \mathbb{R}^2$ in relation to the centroid. To obtain world space coordinates $\vec{p_i}$ of the vertices, rotation and translation are applied to the body space vertices:

$$\vec{p_i} = \begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix} \vec{r_i} + \vec{x} \tag{6.1}$$

previous sections don't use eq numbers. Add (or rm) everywhere!

As the world space vertices are queried often, it has proven beneficial for performance to store the list of world space coordinates of the vertices as attribute and recalculate them whenever the rock is moved. Further dynamic properties include the linear momentum $\vec{P} \in \mathbb{R}^2$ and the angular momentum $L \in \mathbb{R}$.

In addition to the simulated dynamic properties rocks possess, a number of constant attributes ~~that~~ are pre-computed at construction. Most notably, mass and moment of inertia have to be defined for the 2D case. ~~Mass is generally known to be the resistance of an object to a translational force.~~ We define it as is the area of the convex polygon times its density $m = \rho A$. ~~Analogously, the moment of inertia which we use instead of an inertia tensor~~ [10] ~~can be thought of as an object's resistance to torque.~~ Its calculation is based on Fotino's approach [11] and is illustrated in Fig. 9. ~~In body space vertices~~, the polygon is split into counter clockwise triangles $OAB$, with the centroid $O$ constituting the origin. These triangles are subdivided into right triangles. For the inertia $I := I_{poly}$, the following holds:

this is about rock generation

make it a sub-section!

(*)

$$I_{poly} = \sum_i I(OA_iB_i) \tag{6.2}$$

$$\begin{aligned} I(OA_iB_i) &= ccwsgn_{OA_iH_i} \cdot I(OA_iH_i) \\ &\quad + ccwsgn_{OH_iB_i} \cdot I(OH_iB_i) \qquad (6.3) \\ &= \rho \cdot \left( ccwsgn_{OA_iH_i} \cdot \left( \frac{h_i w_{1,i}^3}{4} + \frac{h_i^3 w_{1,i}}{12} \right) \right. \\ &\quad \left. + ccwsgn_{OH_iB_i} \cdot \left( \frac{h_i w_{2,i}^3}{4} + \frac{h_i^3 w_{2,i}}{12} \right) \right) \qquad (6.4) \\ &\qquad (6.5) \end{aligned}$$

you could simply call this $\alpha$!

$ccwsgn$ is 1 if a triangle is counter-clockwise and -1 otherwise.

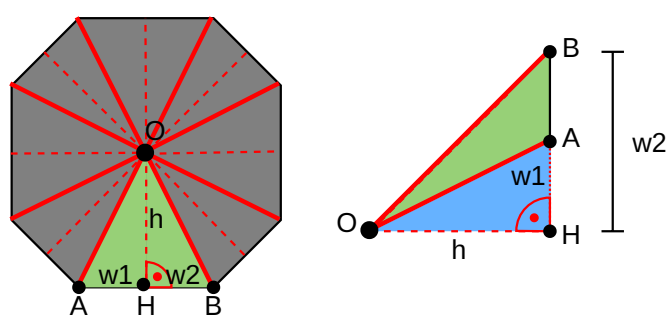



Figure 9: The DynamicConvexPolygon is triangulated. The inertia of the polygon is the sum of the inertia of all right subtriangles. On the right, the blue triangle OAH is clockwise and thus, its inertia has to be subtracted.

gravity alone, now it sounds ambiguous

### 6.5 Movement Equations

The simulation works on the dynamic properties of the polygons. Disregarding collisions, forces such as the gravitational force are solely responsible for the movement of objects. They are applied to the bodies in each step of the physics loop, and the resulting movement is determined. Any force acts on a body in two ways. On the one hand, any

?

You can write this shortly. Use the term body force (g) and surface force



force applied to the body accelerates its linear movement and thus the linear momentum no matter where on the body the force is applied. On the other hand, a force applied to a body also results in a torque $\tau \in \mathbb{R}$, which is dependent on where the force was applied to the body. For a force $\vec{F} \in \mathbb{R}^2$ applied at a point $\vec{p}$ to a body whose centroid lies at the position $\vec{x}$, the toque is

$$\tau = (\vec{p} - \vec{x}) \times \vec{F} \tag{6.6}$$

remove

We define the 2D cross product as

$$\begin{pmatrix} a \\ b \end{pmatrix} \times \begin{pmatrix} c \\ d \end{pmatrix} = ac - bd \in \mathbb{R} \tag{6.7}$$

move this to the top

Put together, the following system of partial differential equations holds for the computation of the movement in two dimensions (c.f. [10]):

$$\frac{\mathbf{d}}{\mathbf{d}t} \begin{pmatrix} \vec{x}(t) \\ \vec{P}(t) \\ \theta(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} \vec{P}(t)/m \\ \vec{F}(t) \\ L(t)/I \\ \tau(t) \end{pmatrix} \tag{6.8}$$

Here, $F(t)$ and $\tau(t)$ are to be read as the sum of all forces or torques at time $t$. We devised a simple, yet effective symplectic Euler solver for this equation:

$$\vec{P}^{t+1} = \vec{P}^t + \delta t \cdot F^t \tag{6.9}$$

$$\vec{x}^{t+1} = \vec{x}^t + \delta t \cdot \frac{P^{t+1}}{m} \tag{6.10}$$

$$L^{t+1} = L^t + \delta t \cdot \tau^t \tag{6.11}$$

$$\theta^{t+1} = \theta t + \delta t \cdot \frac{L^{t+1}}{I} \tag{6.12}$$

let's discuss

Currently, the only force applied in each step is the gravitational force. It is applied at the centroid of the polygons. Additionally, a small percentage of the gravitational force is applied at each vertex, introducing torque in order to approximate air resistance and „real“ falling behaviour.

## 6.6 Collision Detection

After the entities have been moved to their new positions, a check needs to be performed to see whether the new world state is valid. In particular, since the simulation assumes perfectly rigid bodies, all collisions must be resolved. Therefore, for each rock it is checked whether the rock collides with the player, the terrain or any of the other rocks. Because only pairwise collisions are considered and to ensure stability, this loop is repeated until no collisions occur anymore. We do however, allow the simulation some slack by stopping after a maximum number of resolution steps, which is currently configured to 4. This coincides with our design choice of prioritizing performance over accuracy.

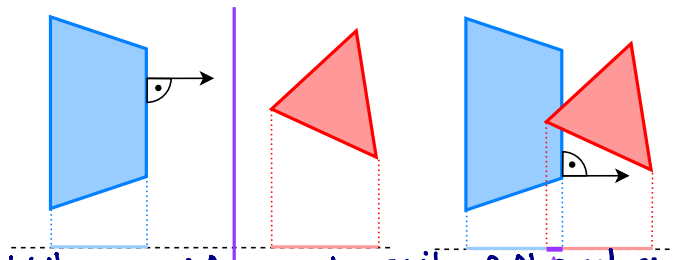
c buffer?

### 6.6.1 Separating Axis Theorem

For collision detection between convex polygons, there exists a plethora of options, the most prominent ones being the Gilbert-Johnnson-Keerthi algorithm (GJK) [12], Minkowski Portal Refinement (MPR) ~~presented by Snethen in~~ [13] and different variations of the separating axis theorem (SAT) [14]. For the sake of simplicity and efficiency, we chose an SAT based approach. The SAT states that two convex polygons do not collide iff there exists a separating axis. If a separating axis exists, a separating axis can be found that is parallel to one of the polygons' edges. Fig. 10 demonstrates this with an example. Our implementation of the SAT roughly follows the algorithms outlined by Wheeler [15]. Separating axes are found by projecting all vertices of both polygons on all edge normals. If a separating axis is found, the algorithm instantly returns. Otherwise, if none of the edges constitutes a separating axis, the edge with the smallest overlap of the projections is returned. This edge is called the reference edge and the corresponding polygon is called *reference polygon*. The other polygon is the *incident polygon*.

if and only if

is this actually needed?



Write new info or describe left and right

Figure 10: SAT: Two polygons do not collide if a separating axis exists. If no axis exists, the algorithm returns the reference edge with the smallest collision depth $\Delta$.

if in need of shortening remove little

### 6.6.2 Optimizations ???

In addition to storing the vertice's world space vertices the, we took other measures to increase the performance of our collision detection routines. Our SAT algorithm increases efficiency by *warm starting*. When a separating axis to another polygon is found, its index is stored in a map accessed by a polygon ID. When those two polygons are checked for collisions the next time, the detection algorithm starts with this edge as there is a high probability that it is a separating axis again and the algorithm can return early.

what is it?

Moreover, we use the concept of *swept axis-aligned bounding boxes* (swept AABBs) [16], [17] (see Fig. 11). The rather expensive SAT algorithm is not executed when the swept AABBs do not collide. In addition to increasing performance, the use of swept AABBs also increases accuracy. If a potential collision is signalled by the AABB check, the last simulation step is substepped if the rocks are „fast“, i.e. the following holds for the two velocities $\vec{v_1} = \vec{P_1}/m_1$ and $\vec{v_2} = \vec{P_2}/m_2$:

$$\|\vec{v_1}\| + \|\vec{v_2}\| > \frac{\text{minPolySize}}{2\delta t} \tag{6.13}$$

We approximate continuous collision detection by using a small but constant $\delta t' < \delta t$. In order to ensure efficient access, the swept AABB is calculated and stored during positioning. Furthermore, to allow local substepping via interpolation, the last position $\vec{x}^{t-1}$ and rotation angle $\theta^{t-1}$ are also kept as attributes.

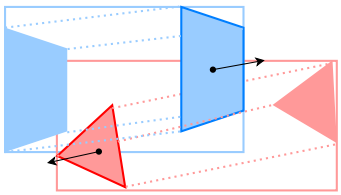


Figure 11: Swept AABBs increase accuracy. Potential Collisions can be detected even if the objects are not currently colliding.

#### 6.6.3 The Terrain

By itself, the SAT only works for convex polygons. However, our terrain is obviously concave. To handle this with our existing collision detection architecture, we triangulate the terrain using the poly2tri library's constrained Delaunay triangulation [18]. Using triangles, all the collision detection methods above work as is. Still, one important aspect to consider is the retrieval of the relevant parts $T_{cd}$ of the terrain and how to create a concave polygon as input for poly2tri from the line representation of the terrain.

In order to achieve this, we first compare the swept axis aligned bounding box $AABB_e$ of the entity with the axis aligned bounding boxes $AABB_{t_i}$ of the terrain sections $T_{i,cd}$. Here, we filter out the sections $T_{min,cd}$ and $T_{max,cd}$ as the first and last terrain sections $T_{i,cd}$ for which $AABB_{t_i}$ overlaps with $AABB_e$. Then, we create a new polyline $Pl_{T,e}$ by combining all sections from $T_{min,cd}$ to $T_{max,cd}$ along the terrain:

$$Pl_{T,e} = (T_{min,cd}, T_{min+1,cd}, \ldots, T_{max-1,cd}, T_{max,cd})$$

After that, we combine the hitboxes $AABB_e$ of the entity and $AABB_{PL_{T,e}}$ of the created polyline into a merged $AABB_{e,PL_{T,e}}$ that includes both of them and has an additional tolerance to each side. Then, we project the first and last point of $Pl_{T,e}$ outwards onto $AABB_{e,PL_{T,e}}$ and add those points $p_{min}$ and $p_{max}$ to the start and end of $Pl_{T,e}$. As a last step, we close the modified Polyline to a polygon $P_{T,e}$ by adding the corners $c_{min}, c_{min+1}, \ldots, c_{max}$ of $AABB_{e,PL_{T,e}}$ in between $p_{max}$ and $p_{min}$ in clockwise order.

$$P_{T,e} = (p_{min}, T_{min,cd}, T_{min+1,cd}, \ldots, T_{max,cd}, p_{max}, c_{min}, c_{min+1} \ldots, c_{max})$$

The process of creating such a polygon for both normal terrain and in case of overhangs is illustrated in Fig. 12.

### 6.7 Collision Resolution

After we found a collision during the game loop, it needs to be resolved. Displacing participants of a collision until they no longer collide restores the validity of the current world state. Applying the appropriate reactive impulses afterwards results in the correct behaviour of entities after a collision occurred.

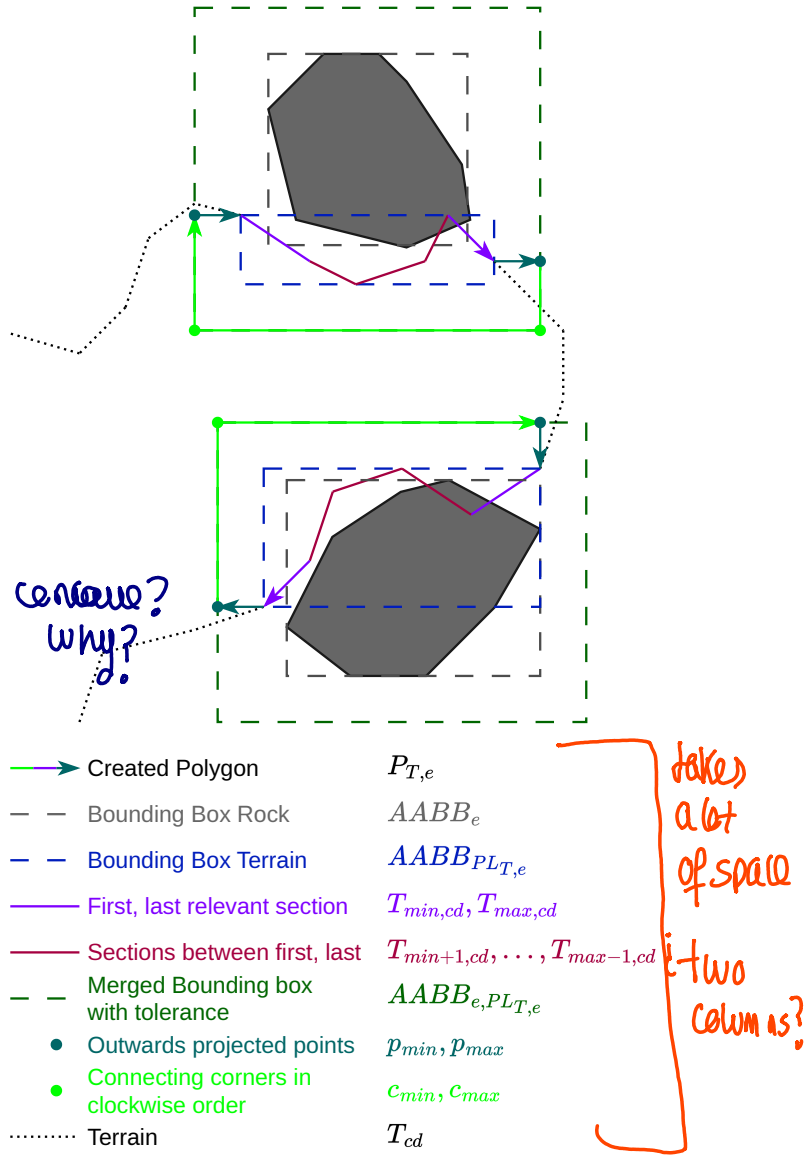



Figure 12: Creating a polygon from relevant terrain sections for collision detection and collision handling with rocks. Here shown for normal terrain and overhang.

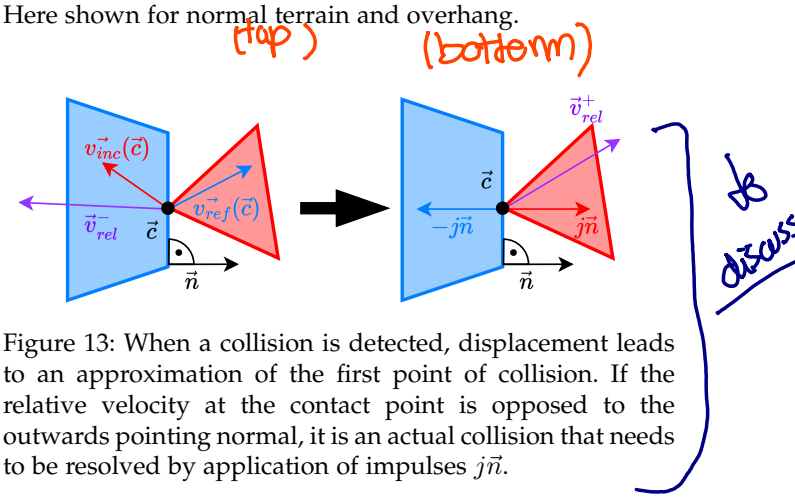



Figure 13: When a collision is detected, displacement leads to an approximation of the first point of collision. If the relative velocity at the contact point is opposed to the outwards pointing normal, it is an actual collision that needs to be resolved by application of impulses $j\vec{n}$.

#### 6.7.1 Displacement

In addition to transferring the world back to a valid state, displacement serves the purpose of approximating the first

* write e.g.) or i.e.)



point of contact between two objects. Only there, impulses can be applied accurately. Once a collision occurs, the colliding polygons are moved by the collision depth $\Delta$ along the collision normal. We weigh the displacement by mass to increase stability [15].

$$\vec{x_{ref}}^{+} = \vec{x_{ref}}^{-} + \Delta \cdot \frac{m_{ref}}{m_{ref} + m_{inc}} \cdot (1 + \epsilon) \tag{6.14}$$

$$\vec{x_{inc}}^{+} = \vec{x_{inc}}^{-} + \Delta \cdot \frac{m_{inc}}{m_{ref} + m_{inc}} \cdot (1 + \epsilon) \tag{6.15}$$

too vague

The superscripts $^-$ and $^+$ denote the state before and after collision resolution. A small $\epsilon$ is introduced here and in other places in the following equations to ensure stability.

If the collision was found by substepping, we first set the polygons to their positions at the time of the collision substep and then apply displacement from there.

### 6.7.2 Impulse Application

Not every contact is necessarily a collision. Only when rocks are moving towards one another at a contact point, we consider it a collision requiring the application of impulses. Fig. 13 shows that this is the case, when the following holds for the relative velocity $\vec{v}_{rel}$ at the contact point $\vec{c}$:

to discuss

$$\langle \vec{n}, \vec{v_{rel}} \rangle = \langle \vec{n}, \vec{v_{inc}}(\vec{c}) - \vec{v_{ref}}(\vec{c}) \rangle < -\epsilon \tag{6.16}$$

The velocity $\vec{v}$ of a point $\vec{p}$ on a dynamic convex polygon can be obtained by the sum of the linear and angular velocity at that point [10]:

$$\vec{v_{poly}}(\vec{p}) = \frac{\vec{P}}{m} + \frac{L}{I} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} (\vec{p} - \vec{x}) \tag{6.17}$$

Finally, the impulse is applied to both the incident and the reference polygon. The formula for the magnitude $j$ of the impulse has been derived in great detail by Baraff:

$$j = \frac{-(1 + \gamma) \langle \vec{n}, \vec{v_{rel}} \rangle}{\Theta_{inc} + \Theta_{ref}} \tag{6.18}$$

with

$$\Theta_{poly} = \frac{1}{m_{poly}} + \left\langle \vec{n}, \left(\frac{1}{I_{poly}}((\vec{c} - \vec{x}_{poly}) \times \vec{n})\right) \times (\vec{c} - \vec{x}_{poly}) \right\rangle \tag{6.19}$$

$\gamma \in [0, 1]$ is the coefficient of restitution. Intuitively, it describes how much of the impulse is preserved and how „bouncy“ a collision is. The resulting impulse is applied as a force $-j\vec{n}$ for the reference polygon and $j\vec{n}$ for the incident polygon at the contact point $\vec{c}$. This force also produces a torque (see equation 6.6). The polygon experiences an instantaneous change in linear and angular momentum.

For terrain collisions, impulse and displacement are evidently only applied to the polygon and not on the terrain triangles which are static.

## 6.8 Beyond Rock Collisions

So far, we only discussed convex polygons. The algorithms as described above can be applied to rocks without restriction. The hiker behaves very similarly but with some key differences. The hiker movement is no longer governed by movement equations, but mainly the user input. For collisions, the hiker hitbox is a convex polygon that does not rotate. When a rock hits the hiker, the rock is destroyed and damage is applied to the hiker. Moreover, a knockback impulse based on the rock's impulse is applied to the hiker. For it to not be overridden by the instantaneous velocity change caused by the user input, the impulse has to be applied with decreasing intensity over multiple time steps afterwards.

Terrain collisions are also different. In air, the hiker bounces off the terrain like a rock would, with a higher coefficient of restitution. On the ground however, the hiker needs to be able to walk smoothly without colliding with the terrain. This is achieved by clamping the hiker on the terrain with a small upward displacement $\epsilon(0, 1)^T$, so that the hiker only collides with overhangs and walls and not the terrain below. Further checks were implemented e.g. to guarantee that the hiker cannot climb or descend a slope that is too steep. ~~These will not be further discussed here.~~ *

The most simple entity is the monster or kill bar. Its movement is only characterized by an x coordinate.

$$x^{t+1}_{\text{monster}} = x^{t}_{\text{monster}} + \delta t \cdot v^{t}_{\text{monster}} \tag{6.20}$$

The texture is always placed on top of the terrain. With the velocity of the monster increasing over time, difficulty is raised. ↓ There are no monster collisions

## 6.9 Performance Aspects

This section supplements the optimizations introduced in 6.6.2 with performance analytics. Thorough scrutiny of the performance of our physics engine is paramount for the game to run smoothly seeing as the physics update is the most important and - if there are many entities - expensive part of the game loop.

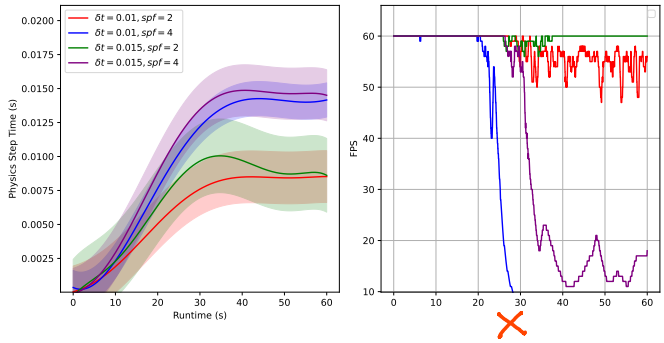
Our physics engine can simulate a heap of up to 50 rocks smoothly and accurately (a heap is the worst case scenario as rocks are constantly intersecting). With the right configuration of the physics engine, the simulation can be pushed to up to 100 rocks, however, some accuracy and smoothness is lost. (*) The crucial parameters for the performance of the physics engine are $\delta t$ and the maximum number of physics steps per frame. The latter presents a solution for what Fiedler [8] calls the „spiral of death“ that happens when the physics engine takes longer to calculate an update than the time $\delta t$ that is simulated. In that case, the accumulator collects more and more time, yet the physics engine cannot keep up, effectively dropping the frame rate to zero. By limiting the maximum number of physics steps that are added to the accumulator per frame ($spf$), this effect can be mitigated to a certain degree. Fig. 14 shows the effects of these parameters on a compute-heavy simulation with many rocks.

Interestingly, the choice of $\delta t$ has only a minor effect on performance. A larger $\delta t$ leaves more room for the physics engine update step and thus the fps drop a bit later. However, the choice of the spf seems far more important. Using an $spf > 2$ we always run into a spiral of death during the simulation. In the actual game however, we even use $spf = 10$. This is because in a real game, the number of rocks is much smaller, so performance is not an issue. The choice of a bigger $spf$ yields more accurate and smooth results. Although the effect of $spf$ on fps is obvious, we

define

(*) how do you measure it? To discuss

can only speculate why $spf$ has an impact on the time needed for a physics update. We assume that this effect stems from stack management overhead when more steps are performed per frame.




also to discuss

Figure 14: Time needed for a physics update (left) and frame rate (right) over the course of a simulation dropping 100 rocks into a bowl in the first 30s. The physics update times are approximated via regression using polynomials of degree 6 and a 95% confidence bound is plotted. For steps per frame spf=4, we run into the spiral of death.

In conclusion, although it is not arbitrary scalable, we deem the performance of the physics engine ~~more than~~ sufficient for the purpose of our game. Using efficient garbage collection when entities and terrain leave the scope of the game in combination with adequate performance allows the game to always run steadily.

## 7 Rendering Architecture

The rendering system of our game is managed by a central `Renderer` class, which oversees all aspects of drawing the game's graphical elements using the Raylib library [19]. This includes not only rendering the world but also managing sub-renderers that handle specific parts of the visual output, such as entities, the terrain, and the Heads-Up Display (HUD) as can be seen in figure 2.

make it a link

### 7.1 Main Renderer Class

The `Renderer` class serves as the core component of the game's rendering pipeline. It is initialized with several key objects, including the world, camera, and resource manager, as well as sub-renderers like the `EntityRenderer`, `MountainRenderer`, and `HudRenderer`. These sub-renderers ensure that rendering is handled in a modular and efficient way, keeping different aspects of the world separate yet cohesive. The primary tasks of the `Renderer` include:

- Managing the camera view and adjusting its position based on the player's movements.
- Rendering the game's background dynamically.
- Delegating the rendering of entities, terrain, and HUD to the respective sub-renderers.

#### *7.1.1 Camera Initialization*

One of the first tasks of the `Renderer` is to initialize the camera. The camera setup takes into account the dimensions of the world, ensuring that the entire world is visible on the screen with the correct zoom level. The following calculations are performed to achieve this:

- The world boundaries are determined based on the game's unit-to-pixel ratio.
- The zoom level is calculated to fit the world width within the screen dimensions.
- The camera's position is centered within the visible portion of the world.

The camera is then updated dynamically during gameplay, adjusting to track the player's movements while maintaining smooth transitions.

#### *7.1.2 Rendering Pipeline*

The `draw()` method is responsible for the main rendering loop. It begins by clearing the screen and then enters the 2D rendering mode with the camera applied. Inside this loop, the `Renderer` draws the background and calls the sub-renderers to draw the game's visual element.

#### *7.1.3 Camera Effects and Fullscreen Support*

In addition to basic rendering, the `Renderer` also handles visual effects such as camera shake, used to simulate rumble when certain events occur (e.g., a large rock falling nearby). The shake effect gradually dampens based on a visual constant and adds a layer of immersion to the gameplay experience.

The `Renderer` also supports fullscreen mode, allowing players to toggle between windowed and fullscreen display. This feature is controlled through the game's configuration settings and cannot be adjusted at runtime, due to visual bugs occuring while toggling.

### 7.2 Sub-Renderers

To ensure a clean and maintainable codebase, the `Renderer` class delegates specific rendering tasks to various sub-renderers, each responsible for a distinct portion of the graphical output.

#### *7.2.1 MountainRenderer*

The `MountainRenderer` class is responsible for rendering the mountain terrain, one of the primary visual components of the game. It uses triangulation to generate a mesh from the terrain data and applies a repeating texture to simulate a continuous mountainous background. This renderer ensures that the terrain is drawn efficiently and updates only when necessary to minimize computational overhead.

7.2.1.1 Texture and Terrain Rendering: The terrain rendering system loads and applies a repeating mountain texture, adjusting dynamically to camera view and terrain changes. The rendering logic involves determining the visible terrain area, checking for boundary or vertex updates, and recalculating the mesh if needed. The terrain is then rendered by applying the texture to the triangulated mesh.

7.2.1.2 Mesh Generation and Triangulation: To generate the mountain surface, the class relies on the terrain's polyline representation. The Renderer can convert the ground points into a series of 2D coordinates, which are then passed to the `poly2tri` library [18] for triangulation. This process creates a set of triangles that form the visual mesh of

you could shorten by avoiding the 7.2.X.Y sections (write as plain text)



the mountain. The generated triangles are stored as vertex data, which are used to draw the mountain with the texture applied.

#### 7.2.2 EntityRenderer

The `EntityRenderer` class is responsible for rendering dynamic game entities, such as hikers, monsters, rocks, and in-game items. It ensures each entity is drawn accurately to the screen, manages animations, and provides debug capabilities for easier visualization of entity states.

7.2.2.1 Entity Rendering: The primary function of this class is to render entities using texture mapping and transformations. Each entity's attributes, such as position, size, rotation, and texture, are considered during rendering, allowing for accurate placement, scaling, and orientation on the screen.

7.2.2.2 Animation Handling: This class also manages animated entities, like moving characters or objects. Animation data, stored within each entity's information, is used to update the current frame of the animation according to the elapsed time. The animation is then rendered frame by frame based on the entity's animation properties, such as the number of frames and frame duration.

7.2.2.3 Hiker and Rock Rendering: Certain entities, such as hikers and rocks, have specialized rendering logic. For example, hikers are rendered differently depending on movement states, including jumping, walking, or crouching. Rocks can be rendered as textured polygons in their normal state, while destroyed rocks involve rendering explosions with more complex visual effects.

#### 7.2.3 HudRenderer

flecs is not the problem if well done

The `HudRenderer` class handles rendering essential HUD elements like the player's health, score, inventory, and altitude, providing real-time feedback during gameplay. It tracks the player's status and environment, updating accordingly.

let's discuss tomorrow!

### 7.3 Debug Mode

The rendering system includes a debug mode that provides developers with additional visual information to aid in debugging and testing. It also removes unnecessary textures that would otherwise clutter the screen. This mode can be toggled on and off, allowing for a clean and unobtrusive gameplay experience when not needed. The debug mode includes the following features:

- **Entity Debugging:** Visualizes entity bounding boxes and collision shapes, making it easier to identify issues with entity positioning and interactions.
- **Mountain Debugging:** Renders the mountain terrain's vertices and triangles, helping developers verify the correctness of the terrain mesh generation.

The debug mode is a valuable tool for identifying rendering issues, verifying game state, and ensuring that the visual elements of the game are functioning as intended.

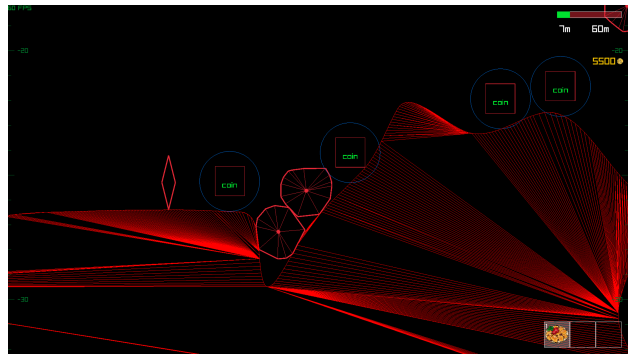



Figure 15: Debug mode visualization of hitboxes and mountain terrain triangulation.

also mention hiker

### 7.4 Optimizations

The renderer optimizes performance by only updating the terrain mesh when necessary, e.g.) ~~such as~~ when boundaries shift or vertices change, avoiding unnecessary recalculations during each frame. Additionally, animations are updated only when the time since the last frame exceeds the animation frame time, and entities that are destroyed and have completed their animations are removed from the rendering cycle.

## 8 Conclusion / Discussion

The original Surviving Sarntal was developed during the Ferienakademie 2023 as a proof of concept under heavy time contraints ~~but~~ suffered from issues with code maintainability, extensibility, and readability due to its reliance on the flecs [4] framework. To address these, we re-implemented the game using ~~object oriented programming~~ {OOP}, which provided greater flexibility and improved code organization. Although adapting from an ~~entity component system~~ {ECS} to OOP required significant refactoring, including redesigning the game loop, the new architecture, while less performant than Flecs, is far more modular and maintainable, easing future development.

hence it

Designing our own physics engine was another pivotal aspect of the rework. The initial implementation handled simple collisions with basic circular objects and a piecewise linear terrain. Our custom physics engine supports arbitrary convex polygons, which allowed us to introduce richer game mechanics and interactions. Though this increased the complexity of collision detection algorithms, our performance benchmarks show the engine is capable of handling a sufficiently large number of entities without significant performance degradation. However, future versions may require optimization for handling larger, more dynamic environments.

In addition to technical improvements, we reworked the terrain system by incorporating two-dimensional splines, offering a more challenging and visually engaging landscape. This change, coupled with the introduction of new mechanics like terrain fall-off and item interaction, has drastically enhanced the gameplay experience. The rework of audio and graphics has further transformed the original idea for the game into an actual enjoyable and immersive game.

change structure

(used in next paragraph)

In conclusion, the rework of Surviving Sarntal resulted in a more modular, maintainable, and engaging game. We applied core software engineering principles and best practices throughout the development process, ensuring high code quality while vastly improving user experience.

## 9 Future Work

While our implementation represents a significant step forward, the project offers ample opportunity for future expansion, including optimization of the physics engine, exploration of new biomes, items and game modes, and introduction of multiplayer capabilities.

controlling?

Therefore, we want to mention some concrete examples and ideas that can be used as starting point for the further development. New items and a leaderboard can be added. A multiplayer mode was proposed, either two users both control their own hiker or one user controls the monster and the other the hiker. The implementation of this mode requires the consideration of several aspects, for instance the potential for player interaction and the resolution of scenarios such as the death of one player. Different biomes with different visuals and terrain generation can be implemented. Another option is to offer difficulty settings to the user and implementing a coin shop for buying upgrades.

Our physics engine also offers various extension points. One known issue with the way our physics are currently implemented is that we assume there are only vertex-face collisions as shown in Fig. 10. However, edge-edge collisions can occur resulting in a jittering effect when rocks are stationary on the ground (see Fig. 16) due to falsely introduced torque. Although this does not constitute a problem for our bouncy physics right now, we are aware that this will need to be solved if the game is to be extended with less bouncy entities. Catto [20] and Wheeler [15] show how to handle this effect using *collision manifolds*. The idea is to consider up to two colliding points between polygons and applying a combined impulse at an average contact point of the two colliding points weighed by their collision depth.

Another way of extending the physics engine would be to add friction. Friction can be modelled as two constants for static and dynamic friction. A friction impulse is then applied during collisions in reverse tangential direction to $\vec{v}_{rel}$. For the magnitude, simply replace the normal $\vec{n}$ with the tangent $\vec{t}$ in equation 6.18. We implemented friction but decided against using it as it would currently negatively impact gameplay, which requires bouncy rocks.

Lastly, one can come up with ways to enhance the performance of the physics engine, e.g. by implementing parallel batch processing and spatial data structures.

To summarize, there are still numerous possibilities for potential improvement and expansion of the game.

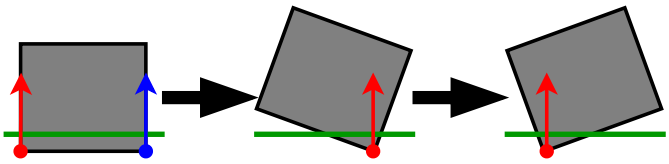


Figure 16: An edge-edge collision should consider more than one contact point in order to accurately resolve the collision. Here, the blue collision point is ignored and jitter is introduced.

## References


[1] K. Schwaber and J. Sutherland, *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game*, october 2020 ed. Scrum.org, 2020. [Online]. Available: https://www.scrumguides.org/scrum-guide.html 1

[2] B. Aheimer, F. Röhr, D. Six, A. Vezenkov, and A. Weckauff, "Project code," https://gitlab-sim.informatik.uni-stuttgart.de/vezenkas/surviving-sarntal, 2024, accessed: 2024-10-13. 1, 2

[3] B. Aheimer, F. Röhr, D. Six, A. Vezenkov, and A. Weckauff, "Project documentation," https://gitlab-sim.informatik.uni-stuttgart.de/vezenkas/surviving-sarntal/-/wikis/home, 2024, accessed: 2024-10-13. 1, 2

[4] S. Mertens, "flecs: A fast entity component system (ecs) for c and c++," 2019, accessed: 2024-10-12. [Online]. Available: https://github.com/SanderMertens/flecs 1, 13

[5] R. Nystrom, *Game Programming Patterns*. Genever Benning, 2014. [Online]. Available: https://gameprogrammingpatterns.com/game-loop.html 4

[6] Unity Technologies, *Unity Manual: Order of Execution for Event Functions*, 2024, version 2022.3. [Online]. Available: https://docs.unity3d.com/Manual/ExecutionOrder.html 4

[7] D. Pflüger, B. Uekermann, and S. Zimmer, *Script for the lecture "Numerische Grundlagen"*, 2024. 5

[8] G. Fiedler. (2004) Fix your timestep. [Online]. Available: https://gafferongames.com/post/fix_your_timestep/ 7, 11

[9] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer, 2008, includes a description of convex hull algorithms like the Monotone Chain Algorithm. [Online]. Available: https://doi.org/10.1007/978-3-540-77974-2 8

[10] D. Baraff, "Physically based modeling: Rigid body simulation," *SIGGRAPH Course Notes, ACM SIGGRAPH*, vol. 2, no. 1, 2001. [Online]. Available: http://graphics.cs.cmu.edu/courses/15-869-F08/lec/14/notesg.pdf 8, 9, 11

[11] R. Fotino. (2020) Moment of inertia algorithm. [Online]. Available: https://fotino.me/moment-of-inertia-algorithm/ 8

[12] E. Gilbert, D. Johnson, and S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," *IEEE Journal on Robotics and Automation*, vol. 4, no. 2, pp. 193–203, 1988. 9

[13] S. Jacobs, *Game programming gems 7*. Charles River Media/Course Technology, 2008. 9

[14] J. O'Rourke, *Computational geometry in C*. Cambridge University Press, 1998. 9

[15] T. A. Wheeler. (2024) 2d collision detection and resolution. [Online]. Available: https://timallanwheeler.com/blog/2024/08/01/2d-collision-detection-and-resolution/ 9, 11, 14

[16] L. Brendan. (2013) Swept aabb collision detection and response. [Online]. Available: https://www.gamedev.net/tutorials/programming/general-and-gameplay-programming/swept-aabb-collision-detection-and-response-r3084/ 9

[17] N. Souto. Video game physics part ii: Collision detection for solid objects. [Online]. Available: https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects 9

[18] "poly2tri: A 2d constrained delaunay triangulation library," 2018, accessed: 2024-10-12. [Online]. Available: https://github.com/jhasse/poly2tri 10, 12

[19] R. Santamaria, "raylib," 2024, accessed: 2024-10-12. [Online]. Available: https://www.raylib.com/ 12

[20] E. Catto, "Contact manifolds," in *Proceedings of the Game Developers Conference*, 2007. [Online]. Available: https://box2d.org/files/ErinCatto_ContactManifolds_GDC2007.pdf 14