you may change the tittle

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

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

Supervisor: Carme Homs-Pons, Institute for Parallel and Distributed Systems

less of FA and more focus in your contributions

Abstract. // **TODO** This project builds up on the outcome of a project that took place in the Ferienakademie 2023 course „Let's play! Simulated Physics for Games“ organised by Prof. Hans-Joachim Bungartz and Prof. Miriam Schulte. The participants of the course created the „Surviving Sarntal“ game, the starting point of this project. „Surviving Sarntal“ is a single-player survival-based game in which the player is a hiker who must navigate a mountainous terrain while avoiding rock fall. The player is provided with the option of utilising either keyboard or gamepad for the purpose of controlling the game. The code is written in C++ and adheres to the object-oriented programming paradigm. The visuals are rendered using the Raylib library.

1 Introduction

this could be longer
you could subsection when you cover flaws of ssarntal v1, goal of the game, input

The objective of the project „Towards real-time physics in a 2D survival-based game“ is a comprehensive restructuring and improvement of the original game „Surviving Sarntal“. This comprises three major work packages: Software engineering, real-time physics and game engineering. The primary objective of the development phase is the enhancement of the real-time physics, which involves the implementation of 2D physics as realistic as possible without compromising real-time performance. First of all, this results in 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. In parallel, the project focuses on adherence to software engineering principles. This includes comprehensive refactoring of the original code base, refining a new and maintainable software architecture, establishing protocols for testing and clean code and creating a platform for detailed documentation for users as well as developers. Furthermore, 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, item functionality and renewed graphics. + proper start screen

do not repeat connectors (eg. at least 10 lines apart)

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.

2 Methodology

you may skip this subsection if you are short in space

2.1 Way of working: Scrum and other methodologies

Prior to starting the project, we, as a team, devised a project charter that encompasses our vision and mission for the upcoming project. Additionally, it includes an analysis

of potential challenges and strategies to address them, as well as concrete working agreements, including tools to use, style guides and organisational decisions. The further development of the game and our collaboration within the team were based on the aforementioned project charter.

On a weekly basis, a meeting was held with our supervisor. During these meetings, the current progress regarding the project timeline, ideas and challenges occurring in development were discussed. Furthermore, two time slots were determined each week for a brief discussion of each team member's current work. The GitLab „Issues“ were used to determine feasible work packages, which then could be assigned to one team member. Additionally, the Gitlab „Issue Board“ was used to provide an overview of the progress of each issue.

Once an issue had been resolved by one team member, a comprehensive review of another team member was required before the changes could be incorporated into the mainline. This measure was implemented with the objective of guaranteeing code quality and correctness.

The project's continuous progress was ensured by structuring it in sprints. We carried out a total of three sprints, whereby at the beginning of each sprint we defined the objectives at the end of the sprint and the issues to be dealt with.

2.2 Installing and Running the Game ~ user guide ~ getting started

* **~~Installing the required Dependencies~~ Installing ~~the~~ Dependencies** In the event that only running the game is desired, the script user_setup.sh 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:

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

⚠ add link to your repo to download

The script is designed to be platform-agnostic, so that it should be operational on the majority of platforms. The following sections provide an overview of additional platform-specific information and instructions for manually installing the required packages.

it's a subsection of dependencies, add a bullet point or indentation

MacOS The installation of the packages requires the use of Homebrew. In the event that Homebrew is not already installed on the user's machine, it can be downloaded from this location (insert link). Alternatively, the user_setup.sh script can be executed to install Homebrew. The following packages must be installed using Homebrew:

```
brew install cmake sdl2
```

On MacOS, it is essential that CMake is able to identify the directory in which the SDL2 installation is located. The CMake script that we have developed employs the following Bash line in order to ascertain the location. It is recommended to execute the following command manually to ascertain the correctness of the SDL2 installation directory.

```
brew info sdl2 | grep 'Cellar/sdl2' | awk '{print $1}'
```

The output should be something along the lines of /opt/homebrew/Cellar/sdl2/2.30.7.

see above MacOS

Ubuntu The execution of the script user_setup.sh includes the installation of the following packages:

- build-essential
- cmake
- xorg-dev
- libsdl2-dev

It is possible to install the packages manually by running the following commands:

```
sudo apt-get update
sudo apt-get install -y build-essential cmake xorg-dev libsdl2-dev
```

Running the Game Once all of the dependencies have been installed, the script run.sh must be executed in order to run the game.

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

that's no longer 2.2

Developing the Game In the event that taking part in the development of the game is desired, the following tools and libraries are needed. Make sure the required dependencies for running the game are installed before continuing.

- clang-tidy
- clang-format
- git
- pre-commit
- python3
- lcov

MacOS The installation of the dependencies requires the use of Homebrew. The dependencies can be installed manually by running the following command:

```
brew install clang-tidy clang-format git pre-commit python3 lcov
```

this seems to be a mistake (repetition from above)

Ubuntu The dependencies can be installed manually by running the following command:

```
sudo apt-get update
sudo apt-get install -y clang-tidy clang-format git pre-commit python3 lcov
```

Note to 2.2 : seems long. worse scenario you refer to website for more details

also no longer 2.2

Testing the Game There are multiple options for testing the game. In order to run the tests with coverage, execute the following command:

```
source build-utils/test.sh
```

In order to run the tests with coverage, the following command must be entered:

```
source build-utils/run_with_cov.sh
```

In order to get a combined coverage report, run the following:

```
source build-utils/test_and_run_with_cov.sh
```

The coverage reports are saved in the directory cmake-build-debug/coverage/reports/index.html and can be displayed with any browser and some IDEs.

not sure this needs a dedicated section, maybe could belong to 2.2

2.3 DevOps

~~Use if~~ YAML configuration ~~in the Project~~ In this project, we employed a YAML configuration file to manage various settings and parameters that control the game's behavior, resources, and environmental constants. The YAML configuration provides a human-readable format, allowing easy modification of game elements such as sound effects, textures, items, landmarks, and game constants without the need for recompilation. This flexibility significantly improves the development workflow by enabling rapid iterations and testing. The configuration is build in the following way.

- Run Mode and Game Settings: The configuration starts with a global setting for running the game in development mode `run-dev-mode`. When this is set to true, the game has some incresed functionality available to developers, allowing to debug specific components more effectively.
- Assets: All audio and visual assets are specified in the configuration.
- Items: Items in the game, such as "kaiserschmarrn", coins, and ducks, are configured with properties like typeId, whether they are automatically collected, and their spawn rates.
- Landmarks: This part of the configuration defines landmarks, such as the Statue of Liberty or Mount Everest, which can be seen as height markers during gameplay.
- Game Constants: This extensive section contains parameters for various in-game mechanics, such as the player's movement, health, and interaction with the environment. Constants for the terrain, item spawning, and physics are also defined here, making it easy to adjust gameplay difficulty or refine specific behaviors without modifying the codebase. For example, the player's jump velocity, friction values, and rock spawn frequencies can all be fine-tuned by adjusting their respective values in this section.

The primary advantage of using a YAML configuration in the project is that it eliminates the need for recompilation when changes are made. This is particularly valuable

I like this part. Think it should be wound up. Maybe 1.2?

during development, where frequent tweaks to game parameters, assets, or settings are necessary. Additionally, the YAML format provides an intuitive, hierarchical structure that makes it easy for both developers and non-technical team members to understand and modify the configuration.

2.4 Quality Assurance

2.5 Transformation of the original game

Entity-Component-System vs. Object-Oriented Programming The original game that was developed in the "Ferienakademie 2023" was based on the Entity-Component-System (ECS) using Flecs. As a team, we decided to alter the project's fundamental architecture from relying on the Entity-Component-System (ECS) to an object-oriented programming (OOP) approach. Given our familiarity with object-oriented programming (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.

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.

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.

3 General Architecture and Modular Design

// TODO add architecture diagram

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. 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.

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.

Services 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 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.

3.4 Rendering Pipeline

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.

3.5 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 controls how and when items and rocks appear in the world based on game difficulty settings.

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

3.6 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.
- **Iterpolator:** Handles the interpolation.
- **Positioner:** Handles the correct positioning of the entities within the game
- **PhysicsEngine:** The central physics engine that manages movement, acceleration, and collision detection. It controls interactions between entities such as the hiker, rocks, and terrain.

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.

so far the architecture part

out of place here I think it fits better in 2.5

3.7 Development Mode

↳ Maybe reorder so that rendering and menus go last.

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.

3.8 Modular Benefits

The modular architecture of the game 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.
- **Scalability:** 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.

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.

if you move this to the introduction (first summarize), then section 3 becomes architecture only :)

4 The Game

should be moved to introduction

4.1 Goal of The Game

The objective of the game is for the player to achieve the highest possible score (measured as the vertical and horizontal distance) by surviving for the longest possible duration while climbing the terrain.

4.2 Entities

Hiker The hiker is controlled by the user of the game and his health is indicated by a health bar. If the hiker loses all his health points or falls behind the kill bar, the hiker dies and the game ends. The hiker has different movement options, including moving to the left and right in the x-direction, jumping and crouching.

Kill Bar/Monster In the current version of the game, the kill bar is represented by a Yeti figure. It is situated at the left edge of the screen and moves continuously while trying to catch the hiker.

Items and Inventory There are items spawning during the course of the game which can be collected by the hiker:

- Kaiserschmarrn: Eating the delicious Kaiserschmarrn will restore the hiker's health.
- Coin: Collecting a coin increases the coin score and increases the speed of the hiker.
- Duck: The duck provides a shield for a few seconds. As long as the hiker has the shield, he won't take damage from being hit by rocks.

The items collected by the hiker are stored in an inventory. The hiker can switch between the items in the inventory and select them for use.

Rocks New rocks are continuously generated and propelled through the air. When the hiker is hit by a rock, the hiker's health decreases.

4.3 The Terrain

this a very important work, could be a standalone section, eg. 4

The terrain the hiker walks on is randomly generated for every game.

Structure Internally, the terrain consists of a list of biomes. Each biome in turn has a ground that consists of a list of points $(p_0, p_2, \ldots, p_n)$. To get a continuous ground, the ground points on the borders of the biomes match. For a reasonably smooth terrain, we decided to use 2D C1-continuous Hermite-Splines to interpolate in between the points. Here, every 2D C1-continuous Hermite-Spline interpolates between a start $p_i = (x_i, y_i)$ and end point $p_{i+1} = (x_{i+1}, y_{i+1})$ given in 2D space. To do this, we use separate 1D C1-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. //**TODO:** Visualization

To get the position of the terrain at a given t we combine the two spline evaluations into a point $(H_x(t), H_y(t))$. We can calculate the C1-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 $H_{i,y}(t)$.

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

maybe add equation numbers?

//**TODO:** visualization
A good overview over hermite splines can be found in //**TODO:** Numerik-Skript?.

We also experimented with C2-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 C1-continuous splines and provide the first order derivatives. These first order derivatives can be freely chosen but need to match at the borders between splines since the splines are C1-continuous. Therefore, we provided the start derivative $(dx_0, dy_0) = (1, 0)$ 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:

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

The individual spline pieces $H_i(t) = (H_{i,x}(t), H_{i,y}(t))$ 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 particularily useful for the generation of the terrain, where the points are generated iteratively using the existing spline representation.

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:

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

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

To improve performance, the ground is approximated by a 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:

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

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

//**TODO:** Collision detection here or later?

move before representation

Generation At the start of the game, the terrain is generated with one start biome of a fixed length. After that, a new biome of the same fixed length is generated whenever the right border of the currently displayed terrain 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. Here, each Terrain phase has the following format:

$ph = ((dx, dy), r, m)$, where:

(dx, dy) : 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 : Number of points to be generated using this phase before selecting a new phase

choose how to write!

To generate a new point p_{i+1} using the existing terrain up to point p_i and a terrain phase ph, we first determine the angular range in which generation of a new point is allowed. Then, we generate a new point within this range and check that it fulfills all constraints. If no such point can be generated, we need to remove some of the already generated points and try to generate the terrain starting from an earlier point.
//**TODO:** algorithm

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 da in beween tried angles is halfed for every unsuccessful try. If

Make constraints a separated subsection after generation

da gets small enough or we excedd 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 here //**TODO:** algorithm. The process is analogous for determining min. //**TODO:** Visualization for min and max-angle, possibly also with inital state and situation where no range can be found

To determine the anglular range and also to verify that the newly generated point is appropriate for the terrain, we formulate several constraints on the terrain:

- **No selfpenetration**: is the spelling correct?
  The terrain is not allowed to intersect with itself.
- **Minimal basepoint distance**:
  The distance between all generated points needs to exceed a fixed lower limit. This limit is not only checked for the basepoints themselves but also interpolated in between the basepoints and checked for the spline representations.
- **Minimal basepoint angle**:
  For every point p_i, the angle between the vectors $\vec{to}_i = p_{i-1} - p_i$ and $\vec{from}_i = p_{i+1} - p_i$ needs to exceed a certain fixed lower limit to prevent the terrain from getting to thin in some places.
- **Hiker clearance**:
  The terrain has to be navigatable for the hiker. This means that the ceiling of an overhang needs to be higher than the hiker is tall at any point. Similarily, the evertical distance between corresponding terrain sections needs to exceed the hiker width in case of pits in the terrain. This constraint is only checked between terrain sections bordered by non-adjacent basepoints to allow overhangs to form.
- **Terrain jumpeable**:
  This constraint limits the height of overhangs and steep terrain. Since the hiker can only climb terrain up to a fixed maximal slope, we need to have a shallower slope in regular intervals. To ensure this, we measure the total height distance of the terrain, where the slope is above the maximal climbeable slope or we have an overhang leading up to the current last point. If this distance exceeds a fixel upper limit, we force a shallower slope.
- **Maximal overhang width**:
  Since a player can only see part of the terrain at any point, they need to be able to see the end of overhangs before entering them. Otherwise, the player might think that these are caves that lead the can exit on the other side. This constraint measures the depth of overhangs and forces the terrain to continue in positive x-Direction if this depth exceeds a fixed upper limit.
- **Minimal x-Pos**:
  To prevent the terrain from genertaing backwards and to prevent cavernous but overall roughly vertical walls, we compute and update a minimal x-Pos that the generation needs to exceed with every generated point.

//**TODO:** Visualizations for every constraint if violated

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 //**TODO:** visualization. If this happens for the first time, we remove a fixed number of points from the terrain. After that, the generation continues as normal. If we cannot regenerate all points that were removed before we retrace again, the we increment how many points need to be removed points by a fixed number and retrace again. //**TODO:** retracing

5 The Game Loop

TODO flow chart of game loop The game loop represents the fundamental structure of the game and is comprised of three distinct phases.

1. Input: Input from the player, whether via keyboard or gamepad, is continuously registered and processed.
2. Processing events and Physics Loop: The game state and game entities are modified based on the player's input and the dynamics of the in-game environment, which are governed by the physics loop.
3. Rendering: The current game state is displayed on the screen.

5.1 Input

- short explanation of libs and how and when processed (maybe howto)

The game permits input from both the keyboard and up to eight gamepads simultaneously. The Raylib library is used for the purpose of input registration. The input handler is responsible for gathering the device events that have been identified by the input device. Once the device events have been gathered, they are mapped to game events and processed by the game event processor, which then adapts the game state accordingly.

again, important enough to have a dedicated section

5.2 Real-Time Rigid Body Physics

repetition? from previous sections

Decoupling of frame rate and physics rate In every step of the game loop, the nested physics loop is executed. This nesting of the physics loop is necessary as the game loop is tightly coupled to the frame rate with rendering being the most expensive task of the game loop. Furthermore, we want our game to run at the same speed regardless of the current framerate. The game should not slow down when the frame rate drops and vice versa. However, to achieve a physics simulation which moves the body in a way that feels consistent and continuous despite framerate drops, we choose to run multiple physics steps between two frames. Each physics step simulates a constant and small amount of time $deltaT$, the size of which can be configured in the configuration. Choosing the deltaT is no trivial manner. If deltaT is bigger than $\frac{1}{fps}$, the physics simulation runs at a slower speed than the renderer. This entails undesirable effects like glitching and improper collision resolution (see below). Although these effects can be mitigated via interpolation, we received an overall better user experience

when running the physics simulation significantly faster than the game loop. The reason for this is that collisions that cannot be resolved in a single physics step are resolved before the next frame is rendered. On the other hand, if we choose a very small deltaT, we encounter two major problems: the computation of the physics step takes more time tha is actually simulated, making the simulation fall behind and if deltaT is very small, rounding off errors might occur. Good values that work for up to 30 rocks at the same time when running the game at 60fps lie between 0.1-0.15s.

The Physics Loop **TODO** flow chart of physics loop To achieve a physics engine running in constant discrete time steps of size deltaT despite varying frame rates, we use the concept of a time accumulator. **TODO** Cite Gaffer "Fix your Timestep" and write our version of the physics loop algorithm here. **TODO** flow chart / pseudocode for physics time step In one time step of the physics engine, updates are executed in the following order:

1. Spawning: Items and rocks are spawned.
2. Event processing: The state of the game and the hiker are adapted based on events that have been created based on user input.
3. Accelerator: The entities are accelerated.
4. Positioner: The positions of entities are updated.
5. Collision detection
6. Collision handling
7. Destructor: An entity will be destroyed if it is out of scope or otherwise undergoes a process of destruction.

Spawning Item Spawning: Items are continuously spawned at random positions above the terrain. The frequency of spawning is dependent on a configurable spawn weight.

Rock Spawning: Different types of rocks (e.g. snow/ ice/ lava/ crystal rock) are continuously spawned and propelled through the air. The specific characteristics of the spawned rock, including its velocity and the frequency of spawning depend on how far the hiker has progressed. The rocks are modelled as convex polygons with the following physical attributes: mass, linear and angular velocity, rotation angle and moment of inertia. 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 (**TODO** do we want pseudocode here?), which is used to determine the convex hull of the set of points. The convex hull represents the vertices of the generated polygon.

No xD

(please add reference to Andrew's)

Destructing

Entity Movement The various entities in the game have different ways of moving. The least complex way of moving is assigned to the monster. Although the monster is

visualized as walking on the mountain, in reality the monster is only characterized by it's x-Coordinate. Since nothing can stop the monster, it is moved in each time step as follows

$$p_{\text{Monster}}^{(t+1)} = p_{\text{Monster}}^{(t)} + dt \cdot v_{\text{Monster}}^{(t)} \qquad , p, v \in \mathbb{R}$$

As game time passes, the monster's velocity is increased, thereby increasing difficulty.

For the rock movement, a more sophisticated approach is used. We decided on basing the rigid body simulation on impulses in order to facilitate collision detection. Rocks are two-dimensional convex polygons. Each rock possesses a position of its centroid $x \in \mathbb{R}^2$, a rotation angle $\theta \in \mathbb{R}$ in radians, defining the angle by which a rock is rotated counter-clockwise around its centroid, and both a linear momentum $P \in \mathbb{R}^2$ and an angular momentum $L \in \mathbb{R}$. We call these rock attributes its dynamic properties and they are put in relation with one another by the following partial differential equation:

TODO Insert movement pde and our discretization thereof (minipage)

Baraff has gone into great detail deriving these equations for the three-dimensional case (**TODO** cite). However, we were able to significantly simplify his equations for the two-dimensional case. An interesting observation is that the rotation in 2D only has one degree of freedom instead of three in the 3D case, because rocks only rotate around their centroid in the two-dimensional plane. As a discrete solver to the aforementioned partial differential equations, we use the symplectic euler method as it is more stable and has a more straightforward implementation. In each physics step, the gravitational force is applied to the rock first, changing the linear and angular momentum, before the position and rotation angle is updated based on the new momentums. The gravitational force is applied at the centroid for a simple change in linear momentum. Furthermore, we apply a fraction of the gravitational force to each vertex of the rock in order to approximate torque, resulting in a more natural way of falling.

The hiker movement stands out from the textbook movement implementation that was used for the rocks. This is due to the fact that the hiker has different modes of movement. While in air, the hiker should behave like a rock that does not rotate. When crouching or moving on the ground however, the hiker is clamped to the terrain and moved with a certain speed factor depending on the slope of the terrain. Also, the hiker should react to user input with an instantaneous change of velocity. To be able to accomodate all these different modes and because of the lack of rotation, we decided on having the hiker based on velocity instead of impulse. **TODO** further explain hiker movement state changes

Collision Detection and Resolution After all entities have been moved, the so-called CollisionHandler service checks entities for collision and handles them accordingly. Different types of collisions can be distinguished depending on the entities participating in a collision. However, all types of collisions have in common, that they

occur between exactly two convex polygons, where the incident polygon participates with a vertex and the reference polygon participates with an edge. Once any collision is detected, a CollisionObject is returned containing additional information about the collision, namely the colliding vertex and edge, the collision normal of the collising edge pointing from the reference to the incident polygon, and the collision depth along the collision normal.
A collision of convex polygons can be detected using the separating axis theorem (SAT) (**TODO** cite) as follows: **TODO** pseudocode of SAT and CD loop and diagram showing an example collision and how it is resolved.

The first type of entity collision is the rock-rock collision. If a collision between two rocks has been detected, the rocks are separated along their collision normal, until they are no longer colliding. In order to achieve this in a reliable and stable manner, we move the rocks along the normal by $collisiondepth \cdot (1 + \epsilon)$. (**TODO** formula) We found an epsilon of 0.01 to work fine. Afterwards, we check if the relative velocity at the collision vertex (**TODO** formula) is greater than zero. If so, an actual collision has occurred, otherwise, the rocks were moving away from one another anyways. If an actual collision occured, impulses from the collision are applied, changing linear and angular momentum. (**TODO** formula) This step of collision detection and resolution is repeated multiple times in order resolve collisions that may have been introduced by resolving other collisions. Nevertheless, it is not feasible to always resolve all collisions before the next physics step. A good value for the maximum number of collision detection steps per physics step has been deemed to lie between 2 and 4. Such a low number works in this case because the deltaT is fairly small with 0.01s per physics step. We cannot afford to perform more collision detection iterations, but we compensate for possible colliding states by performing multiple physics steps per frame.

Furthermore, there are rock-terrain collisions. Calculating with a highly irregular and concave terrain has proven to be a difficult task. We decided on triangulating the terrain using the poly2tri library (**TODO** cite and visualization). To optimize performance, we do not triangulate an entire biome in each collision detection step and for each rock. Instead, we narrow down the terrain to be triangulated by only using sections of the terrain inside the swept bounding box of the rock (**TODO** explain in more detail). When a collision between a rock and one of the triangles of the terrain is detected, it is handled in exactly the same way as with the rock-rock collision with the sole difference being that the displacement and impulse is fully applied to the rock.

- TODO: Terrain collisions
  - Searching with aabb of polygon
  - Creating terrain polygon
  - ...

Another collision type is the rock-hiker collision. If such a collision between the hiker's hitbox (another convex polygon) and the rock is detected, the rock is destroyed, and damage and knockback are applied to the hiker, based on the rock's impulse.

The last type of collision occurs between the hiker and the terrain. Here, many special cases are to be considered depending on the current state of the hiker (**TODO**).

Definition 1. ***Ein Beispiel Konzept*** *Lorem Ipsum dolor irgendwas.*

Eine Citation sieht so aus Barrett et al. [1] zwei Arten von iterativen Methoden.

Definition 2. ***Ein Algorithmus (pls don't delete, I need that as ref)***

Algorithm 1 The Steepest Descent Method

Input: $A \in \mathbb{R}^{n \times n}$, $b, x \in \mathbb{R}^n$, $i_{max} \in \mathbb{N}^+$, $\epsilon \in \mathbb{R}^+$
Output: $x \in \mathbb{R}^n$, $Ax = b'$ mit $\Delta b := b' - b \leq \epsilon \|b\|_2$

```
procedure SteepestDescent(A, b, x, i_max, ε)
    i ← 0;   r ← b − Ax;   τ ← r^T r;   stopCrit ← ε^2 ||b||_2^2
    while (i < i_max and τ > stopCrit) do
        q ← Ar;   α ← τ / (r^T q);   x ← x + αr
        if n > 100 and i mod ⌊√n⌋ = 0 then
            r ← b − Ax
        else
            r = r − αq
        end if
        τ ← r^T r
        i ← i + 1
    end while
    return x
end procedure
```

Corollary 1. *Oh partigano, portami via.*

5.3 Performance

would be nice to provide a table or a plot

The initial implementation only allowed for a small number of rocks to be simulated at the same time. However, we were able to increase performance significantly so that the physics engine is now capable of simulating up to 30 rocks at the same time before the frame rate starts to significantly drop. We deemed this number sufficient as no player could survive such a big number of rock on screen at the same time, and because rocks are quickly destructed once they leave the left side of the screen. Through profiling, we found the most critical performance bottlenecks. Although initially, we saved a polygon's vertices only in body space coordinates around the centroid, we figured that we must also pre-compute the world space coordinates of the vertices in order to avoid having to rotate and translate the vertices each time. Furthermore, the current, last and swept bounding boxes must be kept in memory and updated only once after being repositioned as they are frequently accessed during collision detection. In essence, such computations should be performed only once per physics step after the polygon was

moved but not each time when collisions are checked. Lastly, since we are using an external library for triangulation, we cannot change the time needed for that task. Despite this, we limit the amount of vertices to be triangulated by pre-filtering the terrain against the polygon's swept bounding box.

6 Rendering Architecture

~ The renderen?

chapter looks good but must be summarised!

The rendering system of our game is managed by a central `Renderer` class, which oversees all aspects of drawing the game's graphical elements. This includes not only rendering the game 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).

6.1 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 game 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 game 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.

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 game 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.

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 following steps occur:

- **Background Rendering:** The background is drawn repeatedly to cover the entire game window, ensuring seamless scrolling as the player progresses.
- **Entity and Terrain Rendering:** The `EntityRenderer` and `MountainRenderer` sub-renderers are invoked to draw game entities (such as the player and obstacles) and the mountain terrain, respectively.
- **HUD Rendering:** After exiting the 2D mode, the HUD is rendered by the `HudRenderer`, providing essential information to the player.

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 system also supports fullscreen toggling through a dedicated function that can dynamically switch between windowed and fullscreen modes based on user input (via the F11 key).

6.2 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:

- **EntityRenderer:** This sub-renderer handles the drawing of all dynamic entities in the game, including the hiker, items, and other interactable objects.
- **MountainRenderer:** Responsible for rendering the static mountain terrain and ensuring it is drawn correctly relative to the player's position.
- **HudRenderer:** Focuses on drawing the Heads-Up Display (HUD), which includes elements such as score, player health, and other gameplay-related information.

These sub-renderers are tightly integrated with the main rendering pipeline, ensuring a smooth and visually coherent experience.

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.

Texture and Terrain Rendering Upon initialization, the `MountainRenderer` loads the mountain texture using the `ResourceManager`. The texture is set to repeat using the

`TEXTURE_WRAP_MIRROR_REPEAT` option, which ensures smooth transitions between texture tiles. This texture is applied dynamically during gameplay, adjusting to changes in the camera view or terrain borders. The main rendering logic in `renderMountain()` includes:

- Determining the terrain's visible area based on the camera position and the game's unit-to-pixel ratio.
- Checking if the terrain boundaries or vertices need updating, and if so, recalculating the mesh vertices.
- Rendering the triangulated mesh using the loaded texture, which creates the visual representation of the mountain.

Mesh Generation and Triangulation To generate the mountain surface, the class relies on the terrain's polyline representation. The `updateVertices()` function converts the ground points into a series of 2D coordinates, which are then passed to the `poly2tri` library for triangulation. This process creates a set of triangles that form the visual mesh of the mountain. The generated triangles are stored as vertex data, which are used to draw the mountain with the texture applied.

Debug and Surface Rendering In debug mode, the `MountainRenderer` visualizes the vertices and triangles that form the mountain mesh. The `debugRenderMountain()` method draws red lines connecting the vertices of the triangulated surface, providing a clear view of the mesh structure. Additionally, the renderer can draw a surface outline of the terrain using the `drawSurface()` method, which outlines the ground with green lines.

Optimizations To ensure optimal performance, the renderer only updates the terrain mesh when necessary, such as when the terrain boundaries shift or the vertices change. This prevents unnecessary recalculations during each frame, significantly improving rendering efficiency. Furthermore, the mesh is drawn using OpenGL functions provided by `raylib`, ensuring smooth integration with the rest of the rendering pipeline.

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

Entity Rendering The main function of this class is to render an entity using texture mapping and transformations. The `renderEntity()` function uses the texture information associated with each entity to draw it to the screen. Entities can have various attributes like position, size, rotation, and texture, all of which are considered while rendering. The texture is drawn using the `DrawTexturePro()` function, which allows for accurate placement, scaling, and rotation of the entity on the screen. Debug mode allows the developer to visualize the entity's bounding box using red outlines.

Animation Handling The `EntityRenderer` is also responsible for handling animated entities, such as walking characters or exploding rocks. Animation data is stored as part of each entity's `RenderInformation` object. The `animateEntity()` function is used to animate entities frame by frame, based on the elapsed time and the entity's animation properties, such as the number of frames and the duration of each frame. The current frame of the animation is updated according to the system time and is rendered with `renderAnimation()`.

Hiker and Rock Rendering Special cases like hikers and rocks have customized rendering logic. For instance, the `renderHiker()` function handles different movement states of the hiker: `IN_AIR`, `MOVING`, and `CROUCHED`. When the hiker is moving, it uses the `renderWalkingHiker()` function to either animate the hiker or render the texture based on their current state. Rocks are also handled uniquely, with the ability to render textured polygons for normal rocks and explosions for destroyed rocks using the `polygonRenderer`.

Debug Rendering The `EntityRenderer` offers debug functionalities to visualize entities and their bounding boxes. The `debugRenderEntities()` function renders all entities in a debug mode, highlighting their collision boundaries with lines and circles, and showing labels for items. For example, the hiker's bounding box is rendered in red lines, and the monster's position is marked with a vertical red line.

Destroyed Rocks and Explosions When rocks are destroyed, the class handles explosion animations and renders the resulting debris. The `addExplosion()` function creates a destroyed rock entity with a set animation. These destroyed rocks are stored in a list and are animated until the explosion sequence is complete. The `getDestroyedRocks()` function manages the lifetime of these destroyed rocks, removing them from the list once their animations have finished.

mention all optimizations together as a subsection.

Optimizations To ensure efficient rendering, the class avoids unnecessary recomputation and rendering. For instance, animations are only updated when the time since the last frame exceeds the animation frame time. Entities that are destroyed and have completed their animations are removed from the rendering cycle, ensuring that the game continues running smoothly even with complex interactions between many entities.

State Resetting The `reset()` function clears all animation and destroyed rock data, ensuring that the game or level can restart cleanly without leftover entity states from previous sessions.

HudRenderer The `HudRenderer` class is responsible for rendering the Heads-Up Display (HUD) elements for the player during gameplay. This includes displaying crucial game information such as the player's health, score, inventory, and altitude. It is closely tied to the game world, as it tracks the player's status, environment, and progress. The HUD is essential for conveying real-time feedback to the player.

if short in space it may be removed

Dependencies The `HudRenderer` requires several key game components to function:

- `World`: The current game world, which contains the entities and gameplay state.
- `Camera2D`: The camera used to visualize the game world, allowing for appropriate alignment of HUD elements.
- `GameConstants`: A collection of constants used to configure the game, such as sizes, speeds, and various gameplay parameters.
- `ResourceManager`: Manages the game assets like textures and resources, which are required for rendering HUD elements.

HUD Rendering The main function, `renderHud()`, is used to render the HUD, which displays various UI elements like the score, inventory, health bar, and altimeter. The HUD rendering can be done in two modes: normal or debug, depending on the `newDebugMode` flag. Debug mode allows developers to visualize additional information about the player's status and the environment.

you could summarise a lot here, eg not necessary to explain what a healthbar is. An image would be nice though.

Altimeter The `renderAltimeter()` function displays the player's current altitude in the game. Altitude is a significant element in games that involve climbing or flying, and the altimeter provides feedback on how high the player has climbed or descended. The function also includes `renderAltimeterStep()`, which handles individual steps of the altimeter and adjusts their size and position based on the game's scale.

Inventory and Item Slots `renderInventory()` draws the player's inventory on the screen, providing a visual representation of the items the player is currently carrying. This function works together with `renderItemSlot()`, which displays individual inventory slots. Each slot is drawn at a specified position on the screen, showing the item, if present, in the appropriate slot number.

Health Bar The `renderHealthBar()` function is responsible for displaying the player's health in a visual format. The health bar is a common UI element in games, indicating how much damage a player has taken and how much health remains. This function can visualize health changes, such as when the player takes damage or heals, in real time.

Score Rendering The player's score is displayed using the `renderScore()` and `renderCoinScore()` functions. The score is visualized both as text and as graphical elements like coins or points. The function `drawScoreTexture()` handles the drawing of textures related to the score, while `drawScoreText()` takes care of rendering the numeric score itself.

Landmarks Landmarks are reference points out of the real world, for example the Big Ben. The `landmarks` map is used to store and manage information about these locations. The `loadLandmarks()` function is responsible for loading the landmarks from the game data and rendering them on the HUD, providing progress feedback to the player.

State Reset The `reset()` function clears or resets the HUD state. This is useful when restarting a game or level, ensuring that the HUD elements like scores, health, and inventory are reinitialized and reflect the player's current status.

Debug Mode In debug mode, additional information is rendered to the screen for developers, such as the current state of various HUD components or elements that are not visible during regular gameplay. This mode helps developers test and visualize game mechanics or debug issues related to the HUD display.

this feels like repetition

7 Conclusion / Discussion

here I miss more of comparison with v1.

The project „Towards real-time physics in a 2D survival-based game“ has successfully enhanced the original game „Surviving Sarntal“ in numerous ways. The implementation of realistic 2D physics simulation has resulted in significant advancements regarding the area of game physics while providing the player with real-time performance. In addition to the technical improvements, the project also contributed to an enhancement of the gameplay. The incorporation of novel features and graphics serves to provide the player with an improved gaming experience. Furthermore, the application of software engineering principles has resulted in creating a modular and maintainable code base architecture. In conclusion, the project goals were achieved, resulting in an optimized version of „Surviving Sarntal“.

8 Future Work

this are nice fancy ideas don't forget to mention known bugs too :)

Due to the constraints of time and resources, the development of the game allowed for only a subset of the proposed ideas to be realised. Firstly, it is possible to incorporate additional items into the game. Given that the game has been constructed in a modular fashion, developers are able to easily integrate items with new functionalities. For instance, the idea of implementing a pickaxe that allows the player to destroy rocks by hitting them with the axe was considered. Another proposal is the incorporation of a leaderboard within the game menu, which would display the highest scores achieved by various players. This would include that upon completing a round of the game, the player has the opportunity to save their score under their name. Moreover, the idea was developed that the modular terrain generation could be used to create a variety of biomes, not only differing in their terrain structure, but also in their visuals. For example, we considered creating biomes including an ice, snow, forest or a desert landscape. However, so far, the only biome that has been implemented and is the mountain biome that is used in the current version of the game. Furthermore, the incorporation of a multiplayer mode, which would enable multiple players to engage in simultaneous gameplay, would be very valuable for the users of the game. 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.

Additionally, there are possibilities for enhancing the existing game. One such possibility is the selection of key binds that are more intuitive. Furthermore, the ability to adjust settings, such as the level of difficulty, could also improve the user experience.

In conclusion, there are still numerous possibilities for potential improvement and expansion of the game. Another development sprint could help to realise some of the aforementioned ideas and suggestions for improvement.

not needed, integrate in sections 1 or 2

A Angaben zur Reproduzierbarkeit usw.

Die verwendeten Software-Versionen (Python, NumPy, SciPy, ProbNum), weitere Plots sowie die Implementierungen finden sich in dem folgenden Git-Repository (Read-only access): https://gitfront.io/r/B-Aheimer/voAnN37ECCoS/Iterative-Loeser-fuer-LGS-Seminararbeit/. Für alle Randomisierungen wurde ein Random Number Generator mit festgelegtem Seed bereitgestellt.

References

1. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Donagarra, J., Eijkhout, V., Pozo, R., Romine, C., der Vorst, H.V.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia, PA (1994). https://doi.org/10.1137/1.9781611971538 17

THIS IS WAY TOO LOOOOOOOOOONG!
please make sure that the next draft is close to 12 pages. Final report MUST NOT exceed 12 pages ⚠