Product Requirements Document (PRD)

Team: Engineering Simulations 4
Contributors: Cameron Paul Crutcher, Robert Orlin Jacobson, Levi Z James, Alexander Yang Rhoads
Class: CS46X

Problem Description

Making a game feel good to play requires a lot of trial and error. A developer has to fine-tune the physics of the world and the timing of animations. To do this, they need an environment where they can easily test out the game world, simulating how it works, and getting hard data on the physics parameters that affect everything.

In addition to game development, there is also a need for simulations to be accurate. Accurate simulations are needed in order for companies to test the limits of vehicles and how the forces of our world impact them. By being able to import terrain from the real world and receiving live data from a simulation, engineers are able to build and make decisions around real scenarios with little risk.

Scope

The scope for this project will be limited to the generation of new terrain, the goal is to develop and optimize terrain generation for the Bevy Game Engine, using photographs from the real world to create terrain that interacts properly with the physics represented by the simulation. These will need to have the same sense of collision and physics as the default environment, and the user will need some way to select between environments. With regards to terrain generation, the team is also looking to automatically create environments via the replication of objects using large samples of photos, which is a technique known as photogrammetry.

Use Cases

- A student wants to make a 3D recreation of a park over the summer, but has difficulty drafting it. In order to develop a reference model, they recreate the environment using the simulation, referring to it as they create their own environment.
- A developer wants to configure a baseline gravity for his platformer prototype. The codebase for his game isn't developed enough yet to do it on his own, but he wants to determine the general feel of the game early, so he can take development in a particular direction. To do this, he'll work in the simulation to get a solid idea of what he wants.
- An indie developer is trying to make a small scale game Jam project. She's trying to keep the game small scale, so she decides to develop it in Bevy, using the simulation as a tool to help develop her game.
- The user will drive over the imported terrain, and the collision will work as expected in a real world scenario.
- The user will drive over a certain terrain, then snapshot the data of the forces acting on the car.

Purpose and Vision (Background)

The purpose of the project is to provide an environment where simulated physics can be tested, altered, and recorded. It should be easy to use out of the box, with an environment that allows the user to easily test, and get feedback on, the various aspects of the simulation.

As a tool, it can fit the game development needs for users of all needs and skill. It could be used for a beginner trying to understand the basics, a novice making their first project within the simulation, or an experienced team looking for a basis of data to upscale to their own engine. It

will give them an idea of what to make, data for their physics engine, and even something to make a larger game in.

The photogrammetry feature gives users the ability to explore 3D environments. This has multiple uses. The main use is as a form of reference for environmental generation. The environment being generated will be fully functional in terms of the collision properties between the terrain itself and the vehicle in the simulation. A simulated 3D terrain can be used as the basis or a level, or 3D art. For more niche users, it could serve as a way for hobbyists, like drone enthusiasts, as a way to engage with their interests.

Stakeholders

- Engineering Team
- Project Clients - Weekly meetings with our project partner to stay on track
- Users

Preliminary Context

Assumptions

- We will be developing within the Bevy game engine and will use Rust as our primary programming language.
- We'll be able to use the physics library our supervisor developed.
- A baseline environment where a car can maneuver is a bare minimum. Statistics like speed, acceleration, deceleration, friction, and directional force will need to be displayed and recordable.
- Terrain will be generated using photogrammetry techniques, which will share features of the default environment, including physics and collision.

Constraints

- Time constraints largely may mean we'll have to cut less important features as deadlines near.
- Version 0 won't feature photogrammetry implementation, due to time constraints.
- We'll have to stick to using the limited set of photogrammetry APIs available, as we lack the time and expertise to implement the feature ourselves.
- We'll have to stick to using the Bevy engine and Rust.

Dependencies

- We're dependent on the libraries that are given to us by the supervisors, unless we allocate time to develop a library that matches their function. At the moment, we're using the Rigid Body Dynamics Algorithms physics library by Featherstone. Physics statistics likely revolve around this.
- We'll be using the car simulation developed by our client, Chris Patton, as our baseline simulation.
- We're limited by what is compatible with Bevy.
- Any photogrammetry feature will be dependent on whatever API we use.

Market Assessment and Competition Analysis

Alternatives:

- Unreal, Unity, other game engines: Complicated, and for more advanced users. To accomplish the same things as our simulation, the user will have to understand enough to create their own simulation in-engine
- Godot, Game Maker, RPG Maker, other "starter" game engines: Outside of Godot, most of these can't do 3D easily. All of them are used under the assumption that they'll be for a standalone game, whereas our tool is moreso aiming to assist with specific parts of the development process, not necessarily to be the backbone of a game, unless the user specifically wants that. Our simulation is meant to help work out a game's physics throughout development.
- Games that have an open-ended sandbox/creation mode (Roblox, Dreams): These lack utility as a development tool, especially with regards to fine-tuning/recording physics. It does undercut our project in the sense that a user can use them to sketch a sense of 3D space (where objects go; how it looks to the player).

These are mostly full game engines, so the barrier to entry, as well as speed of use, are higher than our simulation.They also don't undercut our use cases, at least not without a great deal of time and effort on the user's part. In the case of the actual games mentioned, they lack utility as simulators, lacking physics that are either realistic or particularly robust. Using them for terrain generation will also take a longer time on the user's part.

Target Demographics (User Persona)

- Joe is a 19 year old student who wants to make their own game in Godot. He prefers playing around with movement and physics to get an idea of what direction development will take.
- Jane is a 23 year old indie developer making their own game in Unreal. She's trying to figure out a baseline set of physics for her prototype, but lacks an environment to test it out.
- Jerry is a 25 year old engineer that works on making vehicles safer for the public to use.
- Jessica is a 16 year old highschool student taking a physics class and needs help conceptualizing forces.

Requirements

User Story	Feature	Priority	GitHub Issue	Dependencies
As a student, I want to be able to move a character within a 3D environment, as well as interact with it in certain ways, like crashing and falling.	A 3D simulation where the character can move through a larger environment, where these interactions are subject to a set of physics.	Must Have	No	The physics library the car simulation already depends on.
As a developer, I would like to have hard statistics on the object's physics. Things like their running speed, acceleration, height, etc.	A physics statistics tab that tells the user statistics on the object relative to the game world	Must Have	No	The physics library the car simulation already depends on.
As a starting game developer, I would like the ability to both travel through the environment in-engine and noclip through it, so as to better get an idea of how the terrain works.	The act of moving through a solid environment, without any collision detection	Could have	Yes	The currently existing car simulation
As an analyst, I want to watch recorded simulations so that I can learn the effects the previous parameters had on the simulation.	-User inputs and parameters are saved to a local file -Saved simulations can be replayed	-Must Have (local file save) -Could have(recording playback)	Yes	-Workflow pipeline needs to be finalized -Blocked by Engineer user story
As an engineer, I want to import a hill into a simulation, so that I can see the forces acting on a car that will be driving up it.	-Import a mesh into blender -Have the car interact with the mesh properly -Be able to see the forces	Must Have	Yes	-Workflow pipeline needs to be finalized -Integrating meshes from

	acting on the car at a certain moment of time			blender to bevy -Car collision with imported meshes implemented
As a game developer, I want to generate testable environments from my local environment so that I can view which areas I can draw inspiration from for good level design within my own game development project.	-An image processing pipeline where images provided by the user are converted into a testable environment. -A fully functioning car simulation to traverse the environment similar to how an end-user of the final product (the game after public release) will experience the terrain.	Must Have	Yes	-The car simulation needs to be implemented within the project. Terrain physics must be complete before this user story can be completed. 3D environments must have the appropriate generation from images methods enabled and fully functioning. -This user story blocks any final deployments to web infrastructure and optimization for user interactivity with the environment such as responsiveness with the controls.

Non-Functional Requirements

- Input latency should be less than 50 milliseconds (subject to change).
- Maintain 30 frames per second throughout the entire simulation.
- Code should be well-documented and follow the best practices.

- Features, like shortcuts, should have some form of identification in the UI. This can take the form of instructions or a help menu.

Data Requirements

- The project will need to be able to save and load physics, environments, photos, and player settings. These things can exist in a loose folder called, “User Data,” with subfolders for each category of object.
- The project will need to be able to store snapshots of physics statistics.
- The terrain we will be implementing through photogrammetry will likely be of the glTF file format. The Meshroom API itself creates 3D objects as .obj files and a loose set of textures, so it may have to be converted.

Integration Requirements

At the moment, the main point of integration is the Meshroom API. We’ll have to be able to access the API through the simulation, have it generate a terrain using imported photos, and have the simulation make use of all assets Meshroom creates. The terrain may also be edited using the blender API to make the meshes smoother, or more usable for the simulation (Meshroom can create very jagged and inconsistent 3D objects).

User Interaction and Design

At this moment in time, we only know that the simulation will run in a window, with other elements still in the conception phase. Tracking physics statistics will most likely be a UI element on one corner of the screen. Saving and loading terrain, however, is still a bit of an unknown. Ideally we will have dedicated menus, or some call to the OS’s file explorer, but it may even be a command line call if we lack time. It all depends on what we can get done, and it will likely change as we go through versions.

Milestones and Timeline

Version 0 - Dec 10:

- Terrain generated through photogrammetry in Meshroom and Blender (Collision will not be working)
- Terrain generated by rust code with proper collision properties between itself and the car
- Some way to save and load terrain

Version 1 - April 2024

- Physics tracking implemented
- UI elements regarding saving and loading terrain finalized
- Photogrammetry and mesh generation pipeline fully established
- Collision implemented for all loadable meshes

Goals and Success Metrics

Goal	Metric	Baseline	Target	Tracking Method
Basic Collision System (for programmatically generated meshes)	Does the terrain have collision effects? Does that affect the car?	Car doesn't go through terrain.	Cars are affected by things like slope and terrain quality.	Playtesting
Basic Collision System (for photogrammetry objects)	Does the terrain have collision effects? Does that affect the car?	Car doesn't go through terrain.	Cars are affected by things like slope and terrain quality.	Playtesting
Physics Tracking	Are the physics statistics consistently tracked through the UI?	Changes in physics are tracked in UI.	Physics statistics can be saved in a file	Playtesting
Photogrammetry Terrain Generation	Can we use our photogrammetry pipeline to create terrain?	Terrain shows up in simulation.	Regular physics apply to the terrain.	Playtesting

Open Questions

To what extent will we allow user customization? There's a lot of areas where we're not entirely sure to what extent the user should have control. The primary one of these is terrain generation. Using the Meshroom API will often result in weird looking environments, which may be difficult to drive through, or may cause performance issues due to a lack of optimization. To alleviate this, giving the user some degree of freedom in editing terrain could work. It mostly depends on if we can automate optimization of the meshes, and if we have enough time to give the user these options.

Out of Scope

We’re considering integrating some form of web service to the project, such as an online feature that lets users share terrain. This issue is the time constraints, as this feature will only really be developed or implemented should the team finish everything early.